

Introduction to CAT4. Part 3. Semantics.

Andrew Holster

Draft 28 Feb 2021

Abstract

CAT4 is proposed as a general method for representing information, enabling a powerful programming method for large-scale information systems. It enables generalised machine learning, software automation and novel AI capabilities. This is Part 3 of a five-part introduction. The focus here is on explaining the semantic model for CAT4. Points in CAT4 graphs represent *facts*. We introduce all the formal (data) elements used in the classic semantic model: *sense or intension* (1st and 2nd joins), *reference* (3rd join), *functions* (4th join), *time and truth* (logical fields), and *symbolic content* (name/value fields). Concepts are introduced through examples alternating with theoretical discussion. Some concepts are assumed from Part 1 and 2, but key ideas are re-introduced. The purpose is to explain the CAT4 interpretation, and why the data structure and CAT4 axioms have been chosen: to make the semantic model consistent and complete. We start with methods to translate information from database tables into graph DBs and into CAT4. We conclude with a method for translating natural language into CAT4. We conclude with a comparison of the system with an advanced semantic logic, the hyper-intensional logic *TIL*, which also aims to translate NL into a logical calculus. The CAT4 Natural Language Translator is discussed in further detail in Part 4, when we introduce functions more formally. Part 5 discusses software design considerations.

Contents

Part 3. CAT4 Semantics.	3
3.1 CAT2 Semantics: relations and bi-graphs.	3
The first graph reduction.	4
The bi-graph table representation.	5
Useful features of the bi-graph.	6
The CAT2 graph reduction.	8
CAT2 Fact Table.	10
CAT2 graph of joined tables.	11
CAT2 interior category systems.	12
CAT2 logical epistemology.	14
Detail of the second graph reduction.	17
Detail of the table-graph reduction.	19
Appendix 3.1.3 CAT2 graph rules.	23
Appendix 3.1.2 Completeness of 2-graphs mosaics.	25
Appendix 3.1.1 Scalability of RDB, Graph DB and CAT4 single table.	27
3.2 CAT3 Semantics: reference.	35
Reference join concept.	35
Primary points and primary lists.	38
Third join formal structure.	41
CAT3 file-folder hierarchy representations.	43
CAT3 binary relation representation.	44
CAT3 Cartesian Product representation.	46
CAT3 examples of non-relations.	47
CAT3 analogical and recursive representations.	49
CAT3 object part-whole composition.	51
CAT3 class-member relations and negative predicates.	54
CAT3 propositional equivalence.	56
Appendix 3.2.1. CAT3 Propositions and facts.	58
3.3 CAT4 Semantics: time, truth, functions.	60
CAT4 Fact Table.	60
CAT4 Time and dated facts.	61
CAT4 Truth Value.	64
False Facts and Propositions.	65
Propositions in relationship graphs.	67
Symmetric copies.	70
Truth-functions and propositions.	71
CAT4 functions.	73
Summary function examples.	76
Category parents for calculations.	78
Function code and interpreter.	79
CAT4 propositional representation.	81
CAT4 phrase tablet.	83
Phrase compaction and referents.	86

CAT4 Natural Language Translator.	89
Appendix 3.3.1. Semantic completeness and comparison with TIL.	90
Appendix 3.3.2. CAT4 top-down metaphysical categories.	96
References.	100
Acknowledgements.....	101

Part 3. CAT4 Semantics.

In Part 1 and Part 2 (Holster, 2021 a,b) we introduced the CAT2-CAT3-CAT4 relations or graphs mathematically. We now turn to the proposed use, to represent information. We model information as *facts*. Each *point* in the graphs represents a *fact*. These correspond to *propositions* in logic, but they are not quite the same. We can compare with formal semantic logics, which are usually based around propositions and intensionalised on worlds or world-times. But we will start by explaining the representational concept with examples, beginning with the initial graph reductions of *tables* (*relations*) in the next section, and develop the idea from scratch. This was the starting point from which the idea developed. We show how semantic appears in CAT2 first, interpreting relations and bi-graphs. We then introduce the CAT3 system, with *reference join*, and the full CAT4 relation *with time, truth, and recursive functions*. The semantic interpretation is what motivates the formal axioms, so we are most concerned to ensure it makes sense.

3.1 CAT2 Semantics: relations and bi-graphs.

The idea of the CAT2 graph came from first reducing a *table representation* to a simple *bi-graph representation*, and then observing that there is a generalised form of the bi-graph, which is the CAT2 graph, and is a simpler *relation*. This sequence of analysis is illustrated in the first sections below. This gives the main concept, we then move on to interpret this semantically, and design an entire information system around it. As a note for programmers, it may appear at first that the graphs represent a multiplication of data points, but in fact the method turns out to be highly scalable. It is highly scalable for *complexity*, e.g. data representations that would take tens of thousands of tables in a RDB, with unmanageable complexity of joins and functions, are managed automatically in CAT4. In terms of data storage volume, it is similar to an RDB. In terms of querying over data, it is highly scalable. In terms of data integration and exchange between different CAT4 agents, it is highly scalable: any two CAT4 agents can exchange any information about anything, and the integration process is highly automated. These are consequences of the *single table design*.

The first graph reduction.

We start with this basic deconstruction of a simple *table* into a *bi-graph*.

Table for simple relation R .

R	A	B	C	D
a	r_{11}	r_{12}	r_{13}	r_{14}
b	r_{21}	r_{22}	r_{23}	r_{24}
c	r_{31}	r_{32}	r_{33}	r_{34}

Simple bi-graph for R .

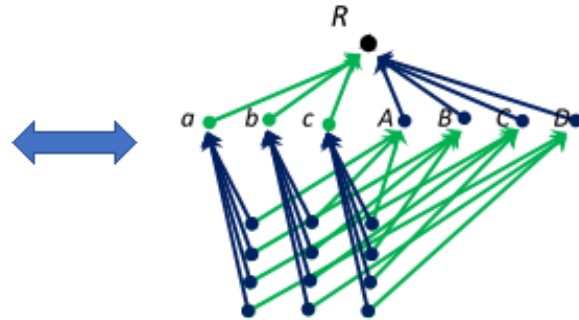


Figure 3.1.1 We deconstruct a simple table, here with 3 rows and 4 columns, into a simple bi-graph. All elements shown in the table-grid ($4 \times 5 = 20$) correspond to points in the graph. The joins represent a generic concept, “belongs to”. The rows and columns belong to the table R , and have one join (out-edge) each. Each cell belongs to a row and a column, and has two joins, a left join to its row, and a right join to its column.

In this simple conventional graph of the table there are four distinct types of points (vertices), for *cells*, *rows*, *columns*, and the *table*, respectively.

- Cell points each have *two joins* (*out-edges away from the points*), one to its row point, one to its column point.
- Row points each have one join, to the table point.
- Column points each have one join, to the table point.
- The table point has no joins.

We see here several types of points being used, matched to four types of entities composing a table.

The bi-graph table representation.

We now examine the bi-graph representation itself, as a relation. Bi-graphs are represented by two tables (classes): one listing all the points (nodes; vertices), one listing all the edges (joins). Points are normally interpreted as ‘entities’ and edges as ‘relationships’ between entities. Edges must also be labelled, as they represent instances of different relationship types. In our first graph, we have four different edge-types, with four different meanings, and these are labelled differently in the edge table below.

Point table		Edge Table			
Point_ID	Name	Edge_ID	Point_ID_1	Point_ID_2	Name
1	R	1	2	1	Column
2	A	2	3	1	Column
3	B	3	4	1	Column
4	C	4	5	1	Column
5	D	5	6	1	Row
6	A	6	7	1	Row
7	B	7	8	1	Row
8	C	8	9	2	Cell_Column
9	$r(1,1)$	9	9	6	Cell-Row
10	$r(1,2)$	10	10	3	Cell_Column
11	$r(1,3)$	11	10	6	Cell-Row
12	$r(1,4)$	12
13	$r(2,1)$
14	$r(2,2)$
...

Figure 3.1.2. Two tables to represent the bi-graph. The *ID* fields are unique identifiers labelled sequentially (with no other meaning). The *Name* fields record names of entities (point table) and relations (edge table). Note the full *point table* in this example has 20 rows, and the *edge table* has $4+7+3*4*2 = 35$ rows.

Now this does not look like a good idea for representing a single table, because we are expanding the nice compact table *R* into a graph that now takes two tables to represent, and it now requires several logical steps (joins between the ID fields) to reconstruct the table relation that we want to see. Graph databases are good for certain purposes, like representing chains of relationships or hierarchical relationships that do not fit easily into table representations, or have domains ranging over diverse classes. But it appears very inefficient to reduce relational tables to general bi-graphs in this way. However, before we give up the thought, we will note some advantages of bi-graphs over relational tables.

Useful features of the bi-graph.

Tables are the standard representational method, and have useful properties, but a database requires many tables, all joined and interrelated and programmed together with functions. This splits the representation into many discrete blocks, each providing a 'data storage' container for facts about a certain kind of entity. Each table also comes with its particular semantics. The problem when it comes to querying data is that it is difficult to join 'facts' together from all the different tables, with their different rules, to get complex and interesting facts back.

The bi-graph gives substantial advantages, which come formally from *reducing the representation in many tables to just two tables, Points and Edges*.

- The bi-graph lets us identify each element of a table individually, including each individual *row*, *column* and cell. Conversely, each element represents an *individual fact* about the table.
- We can extend the *table-graph* by adding points to represent additional information, outside the present table, or table structure. E.g. additional columns; or types of 'meta-data', table joins, file location, properties or relationships, etc.
- We can extend the graph to capture *non-standard relations*, e.g. we can have *two or more cells parenting to the same row-column pair* in the graph representation, but not in the conventional table relation.
- We have missing cells in the graph if values are not populated (taking no space), instead of null cells in the table (taking space and complexity). E.g. this means we can represent *sparse multi-dimensional product spaces* in the graph.
- We can represent as many conventional tables as we like using just the two graph tables. We need just one *point table* and one *edge table* to represent as many tables as we like.
- This graph method is very inefficient for maintenance for one table, because we have to manage *two graph tables*, but with many tables we still require only *two graph tables*.
- Having all rows, columns and cells from many relational tables listed together as points in one point table allows us to freely query over them, and represent relationships between any elements.
- Note *column names* and *tables names* and other 'metadata' contains important information in databases, but we cannot quantify over these in the predominant data-base language (SQL), which is historically limited to *rows* as the first-order variable for quantification.

- The graph also allows us to represent *all relationships of a given entity* around a single point representing that entity, with others in connection. A single point may represent an entity involved in multiple facts.

Hence the graph representation has a representational flexibility that the table does not have. It has better querying scope, ranging across the entire domain of information. We ideally want a method to query over *all facts in our information system*. We would like to be able to quantify over *the domain of all facts*. In a RDB, information is split into a multitude of tables, plus SQL code, metadata, functions, etc, with data entered at different levels of data objects or programming objects. These components all represent information, or facts, of some sort; so how can we even define a *domain of all the facts* represented by a RDB? Graph DB's however allow a single table of all 'entities', and a single table of 'atomic relationships', that can be queried over at once.

In conventional information systems, there is no algorithm to *search system-wide* for facts related to a given subject. It seems almost impossible to define such an algorithm. First you would have to cycle through multiple types of objects, like tables and data sheets, and their properties, metadata, data stores, etc, and examine the information they contain for reference to the subject, and then extract and store that information in some common format. Second, there is generally no formal system to identify the *same objects of reference* across different representations. We have table joins in RDBs, which are formal identities between elements, but generally we must search for subjects by matching on words or symbols (names) – which may or may not correspond to the same referent in different occurrences.

However, *graphs* provide a more general representation we can use to store and relate all kinds of conventional information, e.g. as found in data files or tables or programs, and other more informal sources. We can generally represent an arbitrary *fact* in a graph by adding points for the entities and edges for the relationships it mentions. Graphs also explicitly represent *object identities*. Hence it is possible to build a general representation in a *graph database*, in a way that is not possible in a conventional relational database, with a profusion of tables for every subject.

Graphs centralise and *integrate* information into just *two tables*, which we can readily query. Querying the bi-graph point table is equivalent to *quantifying over first order entities*, and querying the bi-graph edge table is equivalent to *quantifying over first order facts*. But the bi-graph method also has serious limitations, and certainly the method above for rendering relations is not very useful. Instead, we now develop the CAT2 network concept.

The CAT2 graph reduction.

We now analyse the whole table, R , as a uniform graph, which is the standard CAT2 representation.

Table for simple relation R .

R	A	B	C	D
a	r_{11}	r_{12}	r_{13}	r_{14}
b	r_{21}	r_{22}	r_{23}	r_{24}
c	r_{31}	r_{32}	r_{33}	r_{34}

CAT2 graph for R .

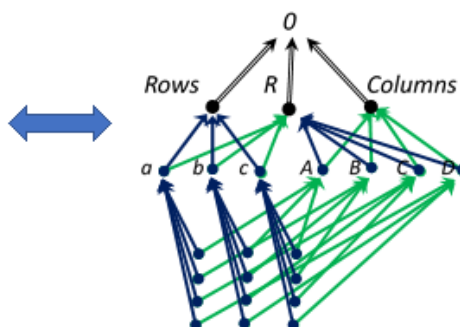


Figure 3.1.3. A CAT2 graph of a simple relation. Note we take the O point to be self-joining, and to be the only self-joining point.

This introduces the concept of a CAT2 relation or CAT2 graph, made uniformly of these simple points-with-two-edges.


- CAT2 graphs are composed entirely of *points each with exactly two out-edges, joining to other points* (i.e. two edges pointing away from the point they belong to).
- Edges are of two types, called *left joins* and *right joins*, labelled with blue and green arrows in the diagrams.

Comparing with the original bi-graph, we now see the distinctive features of CAT2.

- First, every point has exactly two joins (to left and right parents). This applies to the points a, b, c, A, B, C, D , and R . In the original graph of the table, these have one or zero parents. We have made the graph uniform by giving every point exactly two joins.
- Second, we have introduced two new points (not in the original graph), called “Rows” and “Columns”, as parents for the row-points and the column-points. This lets us read each point in the same way, as defining a relationship between the parents mediated by the point-value. E.g. the point a is now read as: “ a is a Row of R ”, and the point A is read as: “ A is a column of R ”, just as e.g. in the previous example we read cell values like: “ $01\ 23456$ is a Phone of Mrs. A ”, etc. Note the left and right joins no longer represent *different empirical relationships*, as in graph databases. Now they represent a general functional relationship.

- Third, we introduce a point at the top, the *zero point* (or *collection point*, or *world point*), which collects all the other points below it. This is the only self-joining point.
- Fourth, we see the *CAT2 flat lattice structure*. The *interior* (parents) of points form flat triangular sheets. This reflects a rule that the left-right parent of a point must be the right-left parent, and this is the natural structure embodied (to first order) in simple tables.

The CAT2 structure can be defined as a mosaic, made from repeated insertions of *points*, each with

two *out-joins*, depicted like: , with *blue and green arrows* representing *left and right joins*.

These are the atoms of *2-graphs* generally. CAT2 graphs are formed from just three types of tilings.

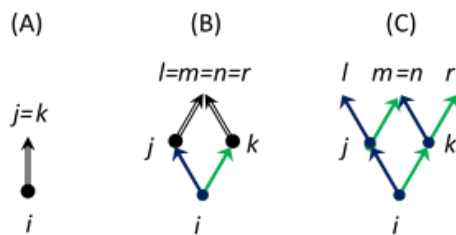


Figure 3.1.4. We are permitted to add a point i to an existing CAT2 graph, containing j, k, l, \dots in the three cases above. Case (A) means we can add a point i with both its joins going to one point, called a *double join*. Case (B) means both parents of i parent to points with double joins, converging to a single point. Case (C) is the case with all single joins in the parent level.

- *2-graphs* made by joining points in just these three *tilings*, without cycles, are CAT2 graphs.

CAT2 Fact Table.

Because CAT2 points have exactly two joins, we can store all the CAT2 graph data in a single table.

We also include a *content field*, to record symbolic data content for points, e.g. a name or value.

ID	ID1	ID2	Name
0	0	0	Zero
1	0	0	Rows
2	0	0	R
3	0	0	Columns
4	1	2	A
5	1	2	B
6	1	2	C
7	2	3	A
8	2	3	B
9	2	3	C
10	2	3	D
11	4	7	r(1,1)
12	4	8	r(1,2)
13	4	9	r(1,3)
14	4	10	r(1,4)
15	5	7	r(2,1)
16	5	8	r(2,2)
17	5	9	r(2,3)
18	5	10	r(2,4)
19	6	7	r(3,1)
20	6	8	r(3,2)
21	6	9	r(3,3)
22	6	10	r(3,4)

Figure 3.1.5. Table representation of the CAT2 graph, with a content field, "Name". All CAT2 facts can be stored in one table. Note the *content field* provides *symbolic referents* for points, which may be references to real-world properties, individuals, etc. We use this to read the meaning of the fact. A *numeric value* is the second content field commonly used.

We will have to extend to CAT3 and then CAT4, with two extra joins, to get a complete system, but this shows the first key feature: all facts can be logically stored in a single table. Equivalently, the class of all facts forms a single first-order relation. This means we can quantify over all facts individually. We can also effectively quantify over *all relations between facts*, in our second-level functions (queries), because all relations are defined by paths or chains joining points, and these are highly structured, with the simple interior topology.

In a real sense, this single table defines the CAT2 *representational space*. But we use *methods* to interpret facts in conventional representations in this space. Now we need to define the CAT2 method more generally than just for representing single *relations*, and next we see the example of *joined tables*.

CAT2 graph of joined tables.

Next we see that CAT2 joins naturally represent *table joins* in relational databases. We consider two joined tables, in a conventional SQL DB.

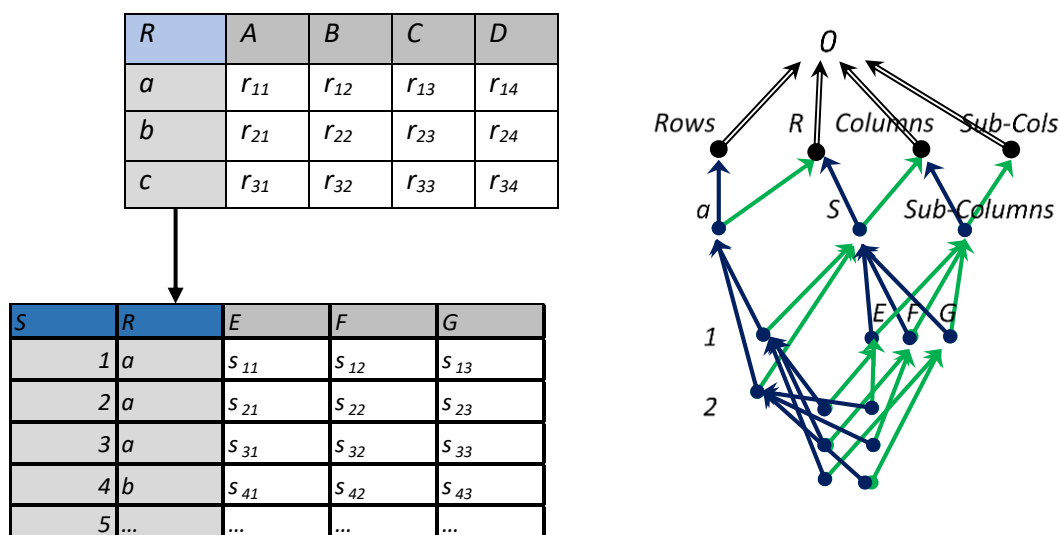


Figure 3.1.6. Left, the secondary table, *S*, is joined 1-many to the primary table *R*. The primary key *R* in *R* joins to the foreign key *R* in table *S*. On the right we see the CAT2 graph, with the *S* data illustrated for object *a* for its rows 1 and 2. This can be applied to the whole tables.

In the RDB content, table *S* is joined (SQL) to table *R*, with the foreign key *R* in *S* joined to the primary key *R* in *R*. The join is 1-many. Tables may have multiple foreign keys and multiple joins to other tables. Relational databases are designed as hierarchical networks of tables, defined with SQL joins.

In the above example, we see the joins between the two RDB tables are represented within the CAT2 graph. The join structure can be extended indefinitely below. So what were represented as table joins in SQL can be represented by exactly the same kind of graph relations as the joins between the cells, rows and columns within a table. We will need to extend CAT2 to make it complete, but this is initially what makes the structure attractive: it provides a seemingly natural relation to represent not just tables but *multiple joined tables*. Thus we will pursue CAT2 as a method for representing a diversity of relational structures in a single complex.

CAT2 interior category systems.

We now jump to the intuitive semantic interpretation, and consider the kinds of CAT2 records we need *above* the fact complexes in the previous examples. As we go up, we require category systems to make sense for the information below.

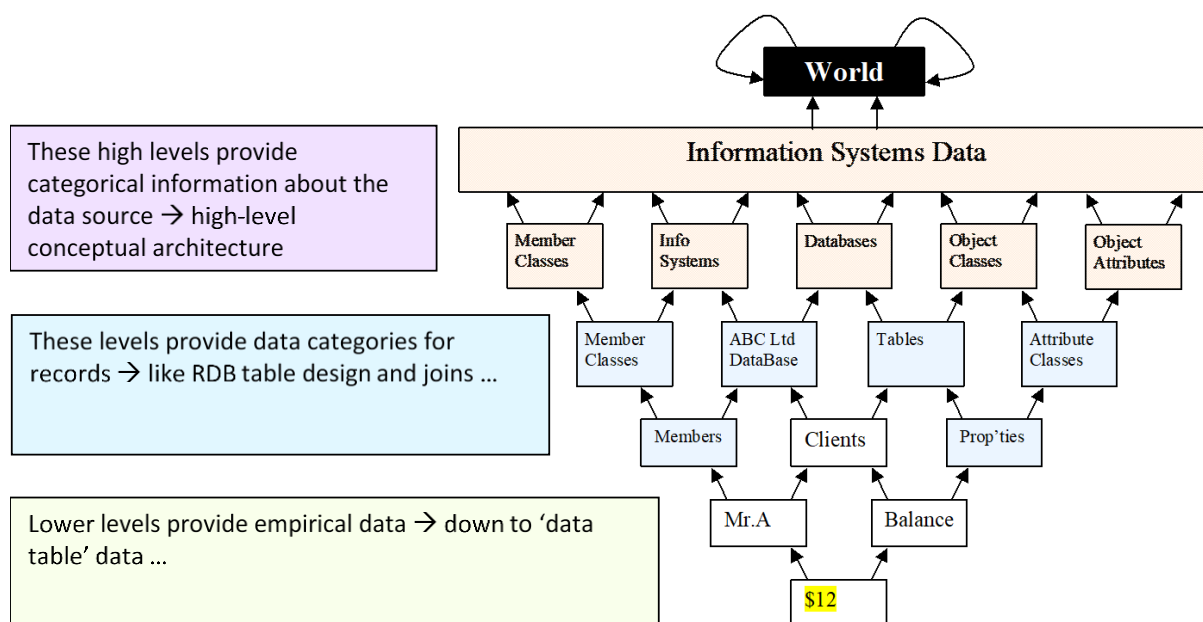


Figure 3.1.7. This illustrates a categorical scheme of a typical type of CAT2 graph. The meaning of the data point “\$12” in this case is found by tracing its parents upwards. The meaning of every point is found the same way: it is a value of the combination of the two parents.

This shows the CAT2 graph extended upwards (the *interior*), putting the elements of the CAT2 table representation into a larger ‘categorical framework’, until we get to a level where we need no more categories. As we go up the classification system, we get to more and more general ‘categorical facts’. The top categories appear somewhat conventional or *a priori* insofar as they are rationalised conceptions of the subject-matter categories. This is what we understand in an intuitive sense as the framework for the ‘meaning’ of facts below.

Intuitively, the meaning of the fact that “Mrs. A is a Member of Clients” *in our representation* also depends on the fact that “Clients is a Table in the ABC Ltd Database” *in our representation*. If the table belonged to another database, the fact would have a different meaning. In turn, we understand the *ABC Ltd Database* to be a database in an IT system, and at this point we stop, and represent this whole bundle of information under a *Collection Point*, “Information Systems Data”, which carries the primary assumption of the *reference to the world*. We can understand

“Information Systems Data” as referring to some domain of real-world facts, with which we are familiar.

In this example, the CAT2 graph is illustrated as representing not only the *information in the tables*, but the properties of the table entities and database entities that provided the original “data containers”. As these software entities are only created to store the primary information, we may drop these layers of *programming information* in a CAT2 representation. We store just the information of interest directly. But this example shows how we can store complex information of several kinds, with data, meta-data, etc, now all in a single relation. Recording the table and database entities and meta-properties is important if we are importing data from a foreign source, and we can use this category structure as an *integration layer*.

Note that the hierarchical categorical structure above, reflecting the *composition of meaning of facts*, now replaces the usual practice of storing different *types of information* across separate data objects, meta-data, spreadsheets, data tables, databases, SQL, programmed functions, etc, which split representations into multiple files and software entities, all with different formats and protocols. CAT2 *integrates diverse information into a single representation*. In practical terms, instead of searching through folders for the right files, tables, subfiles, code, ... to identify a certain type of information, we can search the CAT2 fact table for *any information*. In a complete system, all information and relationships can be transparently represented in this manner.¹

This raises questions of how we decide on the category systems above of course, and this is a very interesting question. But it is not problematic in the first instance. Working category systems are developed in practice just through rationalising practical language use and subject-matter knowledge generally. There is a natural process of *representing meaning* in CAT2.

But at the very top of the tree, we have more metaphysical-type concepts, and we will propose *partitions* of information into some general epistemic categories, such as: *facts, fictions, functions*.

¹ The same goes also for searching for information through object oriented hierarchies of collections, entities and properties, e.g. a screen may be a collection of controls which in turn have types of properties, e.g. “Font Color”, “Data Source”, etc. But this is only a tree. In CAT2, this information can be represented seamlessly in the lattice, with all programming object constructions and meta-data.

CAT2 logical epistemology.

This introduces some very philosophical questions, reflecting semantics and metaphysics. As we go down our graph, into more detailed facts, e.g. from *Members-Clients* to the individual, *Mrs. A*, we are being informed that certain entities exist. These facts represent *empirical category systems* populated with *empirical entities*. At the bottom of the graphs, we end up with the most detailed empirical facts, typically numerical values corresponding to data in empirical data tables, e.g. the “\$12” point. As we go up, we get into more general categories, and eventual a single point. Every point is similar, and the global picture of a CAT2 network around a general point looks like this.

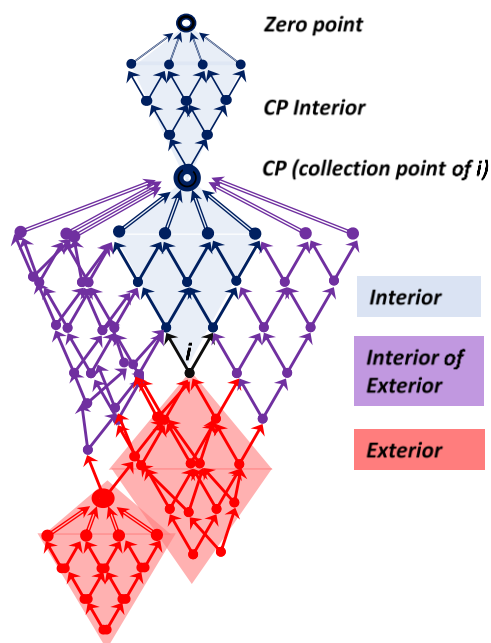


Figure 3.1.8. Illustration of the global structure of a CAT2 graph around a point, i , at the centre. This shows the *interior*(above) and *exterior* (below) of the point i . The interior of i is in blue above i , including i . The exterior is the class of points (red) below i , including i . Indirect relations between facts are evident in the *interior of the exterior* (purple). These points are related to i empirically, or contingently, by *exterior joins* (their meet).

A natural semantics for contingency and necessity in our information model is:

- In CAT2, facts are interpreted as *contingent or necessary relative to each other*.
- Points in the *interior* of a point (parenting points) are *logically necessary to the identity of the point*. Note that *removing a point from the graph means removing all its exterior points*.
- The interior determines the *fact* the point *means or represents*. Hence facts *must be taken to presuppose their parents, grand-parents, etc, i.e. all facts in their interiors*.

- Hence: identity of two facts within one CAT2 or across two different CAT2's requires two points with identical parents, i.e. interiors.
- Facts in the exterior of a point are *contingent relative to that point*. They are not necessary for the existence of the point, rather, they add information about it.
- Facts in neither the interior nor the exterior of a point are neither necessary nor contingent relative to it.

Note that this means that *facts* in our model are not just the *first order propositions* we associate with ordinary statements of facts. We interpret *propositions* later, but note that the same *first-order proposition* may be represented by two distinct points, having the same parents, but with different grandparents. The *first order proposition* is constructed only from the *referents of the three points*. The *fact network* provides a logical background for this proposition, but represents a deeper set of logical relations not evident in the propositional expression.

However we are concerned here with the *contingent-necessary* relation. In conventional philosophy, Empiricists (such as Locke, Hume, Mill, Russell) traditionally hold that facts are of two mutually exclusive types: strictly empirical (synthetic), or strictly logical (analytic). But in CAT2, there appear to be gradations of “*empirical versus categorical*” information, reflecting a natural *relative logical epistemology*. Here is the question for the Empiricists:

- Where is the line in the CAT2 graph between *empirical* and *logical facts*? I.e. between facts representing *logical information* and facts representing *empirical information*?

This reflects a long-standing debate over the categories of epistemology, with the Empiricists dichotomising facts strictly into two classes: *a priori* \equiv *logical* \equiv *analytic* \equiv *necessary*, and: *a posteriori* \equiv *empirical* \equiv *synthetic* \equiv *contingent*. This dichotomy, and the categories of logical epistemology, was most famously contested by Kant *versus* Hume. In the C20th it was challenged by many influential philosophers, such as Quine, Lakatos, Carnap and others, who came to see logical categories and even logical truth as relative, or admitting of degrees, or wholistic, or conventional to some degree. This reflects issues about the relativity and subjectivity of knowledge: because it seems our very capacity for *registering facts* of different kinds depends on contingencies of the linguistic-categorical system (‘conceptual scaffolding’) we have available, as the representational space in which we can formulate facts. This is reflected in CAT2.

- In CAT2, there is no strict division in the graph dividing *empirical facts* (‘*measurements*’, ‘*sense-perceptions*’) from more *conventional facts* (e.g. ‘*classifications of measurements*’).

- In CAT2, a “categorical structure” is required higher in the network as a place to store empirical facts lower down.
- The higher-level categories are represented as *facts* in precisely the same way as the lower.
- The higher-level categories are *not logically analytic* facts, because they are not determined to exist in all CAT2’s.
- They are not simply *empirical*, because they are not verified by simple measurement or observation.
- The higher-level categories are in some degree conventional, reflecting how we like to organise our concepts, and in some degree empirical, required to be adequate to represent the complexity of the more empirical concepts below.

However, although the epistemology of ordinary propositions has become ‘epistemically relative’, note that we can still distinguish the same class, as in ordinary logic, of *purely analytic logical consequences* of a given set of CAT2 facts.

- *Analytic logical consequences of facts* will be represented as facts generated by *functions*, the analytic relation being logically ensured by the *function* implemented.
- The *logical dependency relation* is extended to the 3rd and 4th joins, capturing reference and function.

Logical consequences of existing facts are generated in software by *logical and mathematical functions, applied to existing data*, which generate new *analytic facts*, e.g. facts representing counts, sums, etc. In CAT4, when we insert functions to represent these, we will see that the logical relationships are still represented by the *interior-exterior relations*, but now extended to include the *third and fourth joins*, called the *reference join* and *function join*. Thus ordinary logic of implication, or logical dependency among propositions, will be represented generally by the CAT4 network.

We have briefly seen the larger semantic concept, we now go back to detail of the fact construction, and consider the fine-grained semantics with another graph example.

Detail of the second graph reduction.

The usual method of *graph databases* is to represent *relationships between entities by labelled joins*, like this.

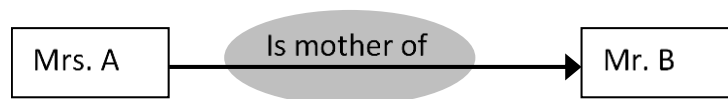


Figure 3.1.9. This illustrates the classic graph technique. Mrs. A and Mr. B are entities, and they have a relationship, viz. *Mrs. A is mother of Mr. B.*

The *fact of this relationship* is normally split into these three components in a bi-graph: two points and one (labelled) edge. This means treating points and edges independently, having multiple edge-types, and allowing any number of edge-types to join points. However another way to arrange the ‘entities’ of the relationship is like this:

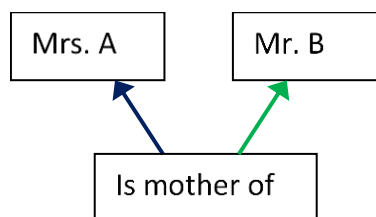


Figure 3.1.10. Edge reduction. This graph means *Mrs. A is the mother of Mr. B.* We have broken the construction down further from the previous graph, into *three points and two joins*.

Now this might seem worse. We now have *three points joined by two edges*, instead of two points joined by one edge, to state just *one fact!* Isn't this inefficient? For space? And for processing the joins? But actually, no: it will provide a means of functional simplification, given that *we can constrain all edges to this simple form*. This again conforms to the concept of a CAT2 relation or CAT2 graph, made uniformly of these simple points-with-two-out-edges. Note that we can use this method to convert all edges in a conventional bi-graph into points-with-two-edges.

- For every entry in the *edge* table: $A\text{--}(E)\text{--}B$, connecting point A to B , we add a new *point*, E , in the point table, and replace the original edge with two edges: $E\text{--}(1)\text{--}A$ and $E\text{--}(2)\text{--}B$.
- The first edge is Type 1 or a *left edge*, the second is Type 2 or a *right edge*. They are both *out-edges of E*.

- If we do this for a complete bi-graph, we end up with a graph defined by a class of points each having either *no out-edges* (the original points), or having exactly one left out-edge (1) and one right out-edge (2).

This gives a graph with just two out-edges per point. This means we can compact the representation, from the two tables for points and edges, into a single table, with each row representing a point with *two joins*. This corresponds to the formal CAT2 table. We still need to impose the CAT2 lattice structure, but this first step, reducing the conventional graph, leads to the same idea as the table reduction. This is because the graph represents classical semantics for classical predicate logic formulae like: $R(a,b)$, while the table represents semantics for the predicate form: $R(a) = b$, and they are both *atomic triadic relations*, relating three entities in a single *fact complex*.

- CAT2 provides a general method to represent relations formed as atomic *triadic relations*. This covers a broad range of representational systems, languages.
- These is a general proposition that all finite graph relations can be defined by constructions from atomic triadic relations; meaning CAT2 in principle can represent all graphs.

The single table representation means that *every edge-fact about in the graph representation is now included as a distinct fact in the point table*, so we can quantify over all the facts directly. *There are exactly two joins per point, and they have a logical meaning*, so we can quantify over all the relationships (joins) directly. We do not have the problem of dealing with multiple edge types having different logics. But what is the logical meaning of the *joins* in CAT2? We continue with another example.

Detail of the table-graph reduction.

Let us now return to our original graph of the table R , above. To make the example more concrete, let us suppose that one of the facts in the table tells us that *Mrs. A has phone number 01 23456*. I.e. let row a be "*Mrs. A*", column A be "*Phone number*", and the cell value be "*01 23456*". Now our initial graph (above) presents relationships like this:

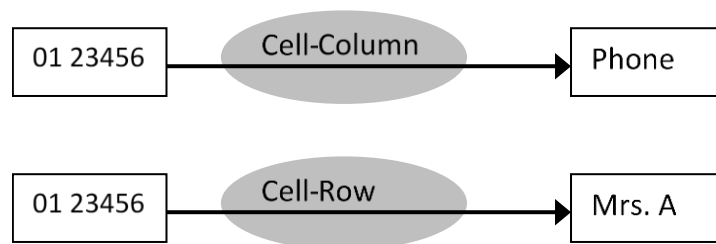


Figure 3.1.11. Conventional bi-graph of relations between table elements.

Using the second method, we would replace these with two distinct point-relations, i.e.

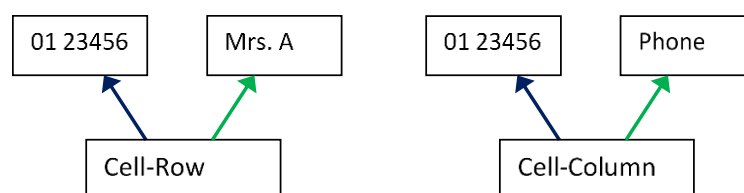


Figure 3.1.12 A second (not very good) way to graph the table relationships.

This contains the information we want, but this is also not very good, because it overcomplicates the information. The *cell-row* and *cell-column* relations are part of the formal table structure, used to represent the fact, but we do not need to include these. Instead we can represent the fact simply like this:

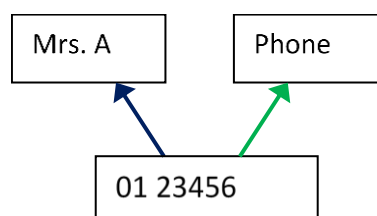


Figure 3.1.13. Representing a *cell value in a table* in a compact C2-type graph.

The left and right joins of the *value point* are now used directly to record the fact that *Mrs. A's Phone is "01 23456"*. We do not have to associate the *values* with *row* and *column collections*, just with the *row* and *column instances*.

The formal representation essentially *just lets us bring entities* - or rather *symbols for entities* - *together in triplets*. The semantics for this fact may be given as follows. *Mrs A is an individual (person)*, *Phone Number is mapping from individuals (persons) to individuals (phone numbers)*, and: *"01 23456" is an individual (phone number)*. We use the symbols (fact tokens) to refer to these external entities, as their *external referents*. We have to worry about this assignment of referents when we read the symbols, to interpret its real-world meaning. But as far as the formal representation goes, all we have to do is store the three symbols, the *fact tokens*, in relative order to each other. The arrangement above is *read with the interpreted semantics*. CAT2 does not tell us how to interpret the semantics, i.e. entities or types of entities referred to. It just represents a framework of *hierarchical functions*.

Given we have seen several ways of representing information, we may ask: is there an optimal way to use CAT2? Does the interpretation of data in CAT2 conform to general rules? Is there a general method for reading CAT2 data? We will see that although there are always several ways to store individual items of information, collections of information are naturally stored in uniform structures, there are general rules for reading information and hence writing information, and the rules reflect general semantic concepts.

To illustrate the role of conventional semantic intuitions further, we can use the same formal translation as above of any *cell value* from a table into a conventional bi-graph. But this *method* does not immediately conform to our semantic intuitions. E.g. applied to a table of values, e.g. *phone numbers of persons*, this would correspond to a conventional bi-graph like this:

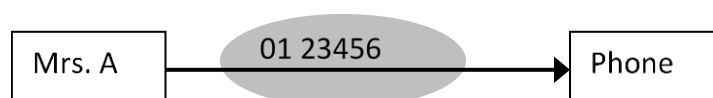


Figure 3.1.14. A conventional graph of a 'relationship', labelled "01 23456", between two entities, *Mrs.A* and *Phone Number*.

We do not normally make graphs like this – because it makes the phone number "01 23456" appear as a *relationship between Mrs. A and the Phone Number property*, and we do not normally think of values in this way, as *relationships*. This 'relationship' labelled "01 23456", is likely to be unique to Mrs. A, making it suitable to represent as a (slowly changing) property-value in a table, whereas the

use of graph databases is usually to record relationships that have many instances, and form networks relating extensive groups. Thus we usually interpret bi-graphs and database tables semantically, in terms of the expected *structures* of the relations. If we get it wrong, the data will not fit into the ‘data containers’ we design.

But our point of view here is more abstract. All data fits into the (single) data container. We only care about the *formal relationships between data elements*. CAT2 formally represents ‘atomic facts’ as always relating three entities at a time: *the point-entity and the two parent-entities*. These are typically conceptualised as: *object-property-values*, as in a typical table of property values, but may equally be used to represent: *object-object-relations*, as in a typical relationship table, such as the bi-graph edge table. But it does not matter to us what content CAT2 is applied to, formally it is all just about functional mappings for us.

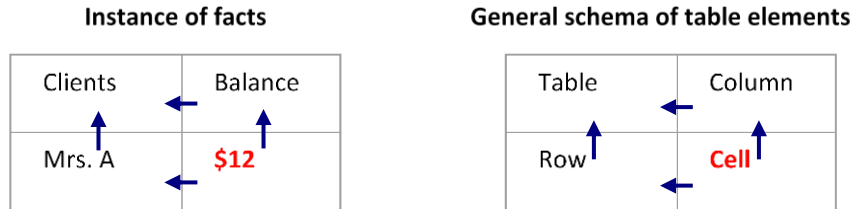
CAT2 provides a hierarchical network of functions. The structure matches that of typed hierarchical functions in higher-order logic. Our ‘type hierarchy’ is represented by positions in the *interior and exterior coordinate structure*. General functional application, of p and q together, is defined by the general function: $F(p,q) = p \vee q$. This works to produce the *application of any conventional function p to argument q* . But it is formally symmetric: there is no difference between ‘applying p to q ’ and ‘applying q to p ’. It is more general than the *first-order functional application*, which is only the case where p and q are neighbours in CAT2. I.e. when: q is in either: $p.[1,-1]$ or $p.[-1,1]$. The entire class of binary functions of p includes the array of mappings: $p \vee p.[x,y]$, where x,y are finite integers, bounded by the depth of the exterior and interior of p .

In any case, our use of the CAT2 graph introduces an unusual shift in semantic theory for databases. We have identified a single type of hierarchical relation for the construction of all types of facts: but as illustrated next, this forces us to revise the concept of fixed grammatical roles of terms.

Table semantics may be analysed as relations between symbolic elements.

We have a table *C*, row *A*, column *B*, and cell *f*. This asserts: *f* is the *B* of *A* in *C*.

E.g. *\$12* is the *Balance* of *Mrs. A* in *Clients*.



The arrows show *deterministic functions*.

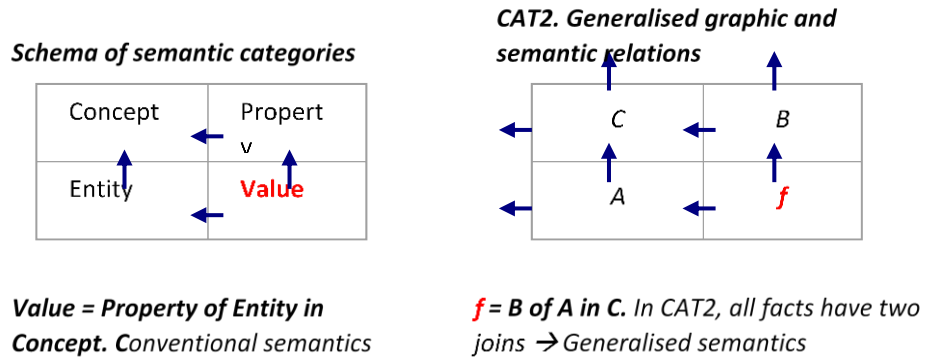


Figure 3.1.15. Generalising the semantic schema. CAT2 applies the same functional schema to read any data. Terms that act as *predicate terms* naming *properties* in one fact, may act as *subject terms* naming *individuals* in another fact.

The CAT2 claim is that we can represent and read any form of data, or *facts*, essentially in the form of this schema, applied to a CAT2 graph with any point chosen as *f*. This means the grammatical roles of terms are not fixed, rather, they are relative, and vary according to the logical construction.

The line of thought we have followed was essentially the origin of the concept, and it leaves us with the problem now of interpreting and extending the semantic concept to make it adequate, and applying it to more examples, to demonstrate its practicality and universality.

Appendix 3.1.3 CAT2 graph rules.

A full set of rules is proposed in Part 1 (CAT4 Axioms), but we restate the key CAT2 rules more intuitively here for convenience, before extending to CAT3 in the next section.

CAT2 graphs may be considered as mosaics, made from uniform *points*, each with two *joins* (*out-edges*). We can call the general class *C2-graphs*. The class of all possible C2-graphs is quite large, and has little structure. We pick out the special CAT2 structure by adding further rules. This forms the class of CAT2 graphs, which we may regard as our *representational tokens*. The two key rules for the CAT2 structure are:

- *The Closure Rule*: Every path of joins goes to a unique point at the top (no cycles), and:
- *The Diamond Rule*: The left-right parent is generally the same as the right-left parent

We refine the second rule:

- A point can be *double-joined*, meaning both its joins both go to the same point, or *single-joined*, meaning its joins go to two separate points.
- Double-joined points can be double-joined to any other point.
- Single-joined points must be joined to obey the Diamond Rule.
- Each point must join exclusively to double-joined points, or exclusively to single-joined points, not to a mixture.

The reason for the last rule is to make transitions in the network from double-joins to single-joins smooth and easier to manage. These are transitions between tree and lattice-like sub-structures. The last rule endures these are smooth rather than ragged, making the functions easier.

This illustrates the rules for inserting points.

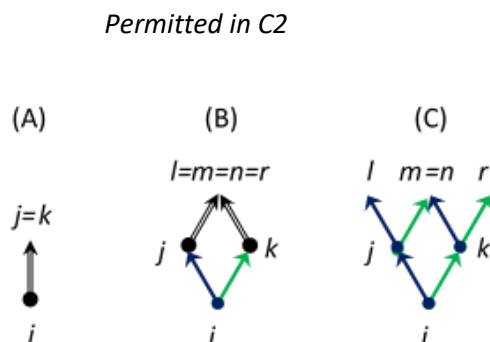


Figure 3.1.16. We are permitted to add a point i in each of the three cases above.

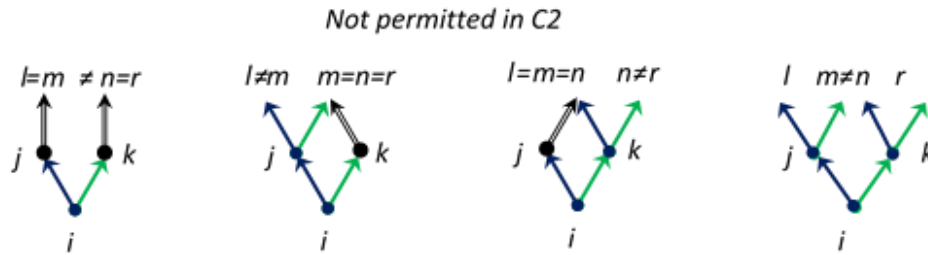


Figure 3.1.17. We are not permitted to add the point i in any of the four cases above.

There is one further rule, which is the Zero Point rule. The simplest CAT2 graph is *a single point, double-joined to itself, i.e. the zero point alone*. We can build all CAT2 graphs recursively, starting with the zero point, and adding new points, in any pattern allowed above. These are the essential rules to recursively define the CAT2 networks. The fact we need only refer to the immediate *parents and grandparents* means it is easy to check these insertion rules *locally*. It is shown in the axiomatisation how these local rules result in the large-scale topological properties.

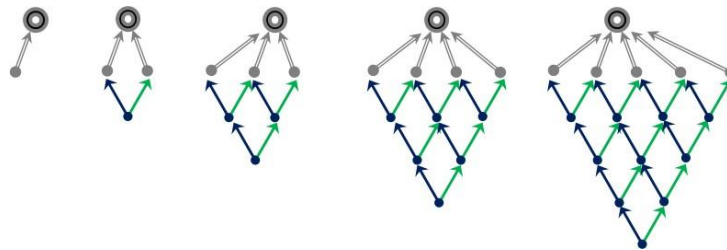


Figure 3.1.18. These show the *interior points* of the points at the bottom of each graph. The *interior points* in CAT2 always forms a lattice pattern like one these (to any finite depth). The points at the top are called the *collections points* or *CPs*.

Interior graphs always collect to a point, called the *Collection Point*. This pattern is determined by the CAT2 graph rules. A CAT2 graph consists of multiple layers of such *flat lattices*. This allows a positional coordinate system, which determines positions of point-intersections.

The point-edges can now be represented more compactly than the general graphs, in a single table, with one row representing each point and its two joins. This is now a highly constrained relation. In particular, the CAT2 *interior network structure* is very simple, it conforms to a simple topology, a positional coordinate system, and this becomes the single primary network structure we have to program logical operations for.

Appendix 3.1.2 Completeness of 2-graphs mosaics.

We will need to show our representation is complete in certain respects, which it is not yet. However there is an important consideration to start with, which is that the graph mosaic we are using must be capable in principle of representing any other type of bi-graph. We can define *bi-graphs as mosaics*, formed from points with a certain number of *out-joins*. The number of joins per point in a bi-graph is usually variable, but we can define *n-graphs* (or graph mosaics) as made purely of points with a fixed number of out-joins. We call these *points-plus-n-joins* the *n-atoms*.

- 1-graphs are graph mosaics formed by points with one out-join (1-atoms; 1-tiles).
- 2-graphs are graph mosaics formed by points with two out-joins (2-atoms; 2-tiles).
- 3-graphs are graph mosaics formed by points with three out-joins. (Etc).
- Any graph can be used as a *tile* to create a new type of graph mosaic.

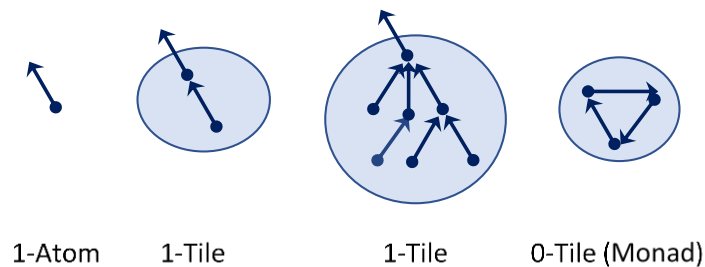


Figure 3.1.19. With *1-atoms* we can only make graphs representing *trees*, with one out-edge, or *cyclic-trees*, with no out-edges. These in turn can only combine to make further 1-tiles.

We cannot make a 2-tile or 2-graphs from 1-atoms. Hence *1-atoms have a severe limitation* as a basis for a representation. We need points with at least two out-joins, i.e. *at least 2-graphs*.

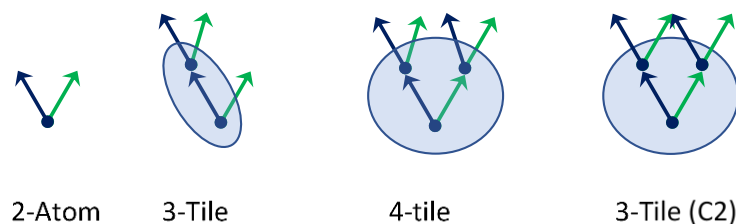


Figure 3.1.20. With *2-atoms* we can make graphs with 3 or 4 out-edges (or indeed any number), which can be used as *3-tiles* or *4-tiles*, etc, which can be used to make 2-graphs that are isomorphic to any 3-graphs, 4-graphs, etc.

We can therefore define tiles from 2-graphs with three out-edges, and use these as if they were 3-atoms, to build 3-graphs over a basis of 2-atoms.

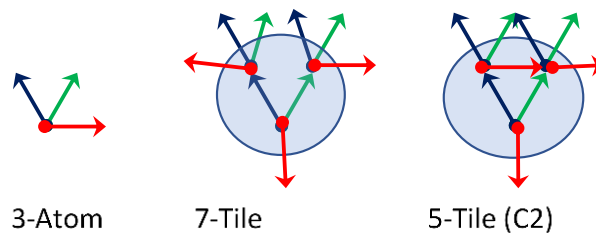


Figure 3.1.21. 3-atoms can make graphs or tiles with any number of out-edges.

The key point is that *1-graphs* are not sufficiently complex to represent fact networks, while *2-graphs* can represent any network, including *3-graphs*, *4-graphs*, etc. Applied to a symbolic representation, we call this the “*Triadic Principle*”.

- For an adequate system of symbolic representation, *atomic symbols* (words; points) in *expressions* (sentences; graphs) must be able to associate at least three entities, or symbols for entities, in complexes at once.

Note points in *1-graphs* associate just two entities at once (the point and its one parent), *2-graphs* associate three entities (the point and its two parents), etc. For a proof of the limitation of *1-graphs*, show that there are only two types of (connected) *1-graphs*: an *open 1-graph* has exactly *one open (unassigned) join*, and a *closed 1-graph* has *no open (unassigned) joins*, and exactly one cycle. This is evident because the total number of *joins* in a connected *1-graph* with *N* points is *N*, and at least *N-1* joins must be *closed* for the graph to connect *N* points, leaving one join that may be closed or open. Or for a recursive proof: start with a *1-graph*, add a *1-atom*, prove it has the same number of out-edges; prove adding a *1-atom* to any graph leaves the same number of out-edge.

- If a *1-graph* has no cycle it is a tree, and the root is the point with the open join.
- If a *1-graph* has a cycle, it is equivalent to a set of *in-trees* rooted to any points in the cycle.
- An open *1-graph* (tree) can be made into a closed cyclic graph by assigning its open join to any point in the tree.

Thus any connected *1-graph* has at most one open join, available to relate it to another *1-graph*. There is no way to *relate 1-graph complexes to each other in any different topology than the atomic tree relations*. So the structure of *1-graphs* is very limited. By comparison, *2-graphs* can represent any *n-graph* structure, through homeomorphisms to sub-graphs. This means *2-graphs* are adequate in principle to represent any complex networks. However we must extend CAT2 to make it adequate for a semantic model, and we now continue with this.

Appendix 3.1.1 Scalability of RDB, Graph DB and CAT4 single table.

We contrast the methods of *relational databases*, *graph databases*, and *CAT2*, in terms of the layers of data objects and code, which limits the feasible algorithms or automation, and scalability of the systems. Coloured symbols in these diagrams indicate *hard-coded contingent objects*, like tables, queries functions, etc. These are designed and programmed to capture specific concepts. The grey symbols signify generalised logical or system objects, like logical tables or logical processes, which are independent of data-content, and hence are more universal algorithms of the system.

Relational Database: Many Tables.

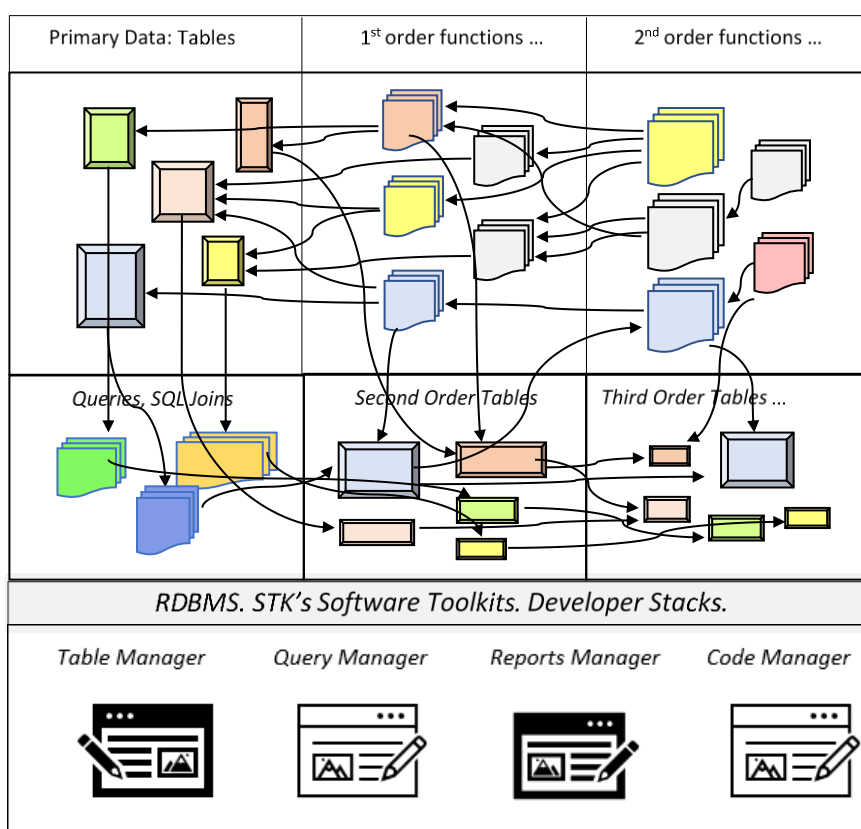


Figure 3.1.22. RDBs have many *primary tables*, each table *hard-coding* the equivalent of an *empirical predicate language*, one term at a time, in *tables* and *columns*, and wiring all their logical relationships together, with SQL joins and 1st order functions. We need a layer of 2nd order functions to manage this layer, with 2nd and 3rd order tables.

RDBs lead to complex *ad hoc* networks of *ad hoc* tables, maintained with joins and functions. To organise the layers of objects and code underlying the construction we have *RDBMS's*, i.e. *Relational Database Management Systems*. These provide third order systems to manage second order code to manage first order data. Note the data representation is also complicated by all the layers of

software objects required to represent the fact data. E.g. tables have columns each with properties like data type, format, size, validation rules, default values, indexes, keys, etc. The RDBMS supports all this meta-information – as a layer for managing the information required to create the representational objects to manage the factual information. But important information is now also scattered through the meta-data level, e.g. the table joins, column names, functions, etc. The system becomes unscalable as we extend the information domain, and increase the number of primary tables, because their relationships and processes multiply, and must all be manually coded.

Graph DB.

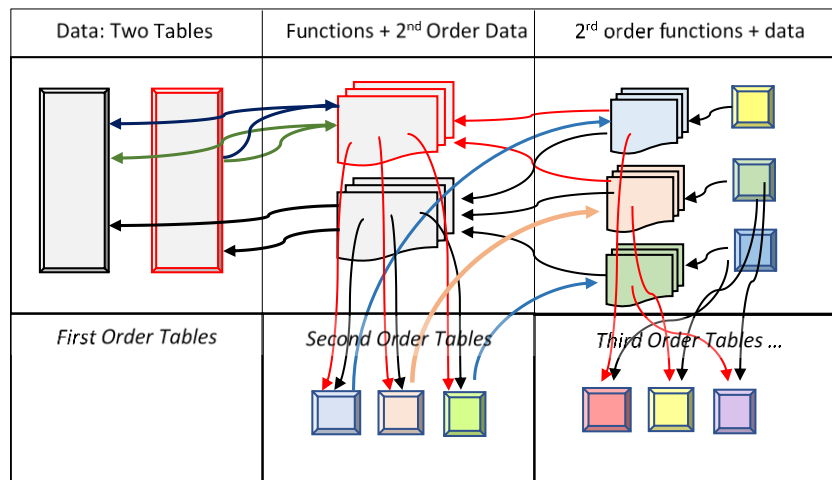


Figure 3.1.23. The bi-graph representation is more compact than the RDB, with two primary tables, for points and edges. However the complexity re-emerges at the second order.

This illustrates that the bi-graph representation is relatively simple, with just two fundamental tables, and in important respects this is more universal and flexible than RDBs. However the programming complexity now re-emerges in managing the edge structures.

We should emphasise that this method is very inefficient for storing relational data if we use the: $point \rightarrow (relationship) \rightarrow point$ model, with two facts defining a table cell-value, e.g.: $\$12 \rightarrow (column) \rightarrow Balance$ and: $\$12 \rightarrow (row) \rightarrow Mrs$. This is not a viable method for bi-graphs, as observed earlier. It represents redundant facts about the table representation in the relevant facts about the entities. But we can put edges directly connecting: $\$12 \rightarrow (Balance) \rightarrow Mrs A$. Or alternatively: $\$12 \rightarrow (Mrs. A) \rightarrow Balance$. This is a possible method, but it just leads to the hierarchical database, with tree structures and pointers. This has limitations such as Codd [1970] identified for large primary relations, where Entity-Attribute pairs are saturated with single n-tuples. But graphs have advantages for storing 'sparse data', representing many attributes across many entities at once.

The main use of graph db's so far is not for primary data storage (as in on-line transactional data systems), but for analytics, analysing and comparing hierarchical-type relations and network relations, with domains of many individuals and diverse relationships and attributes. The RDB is poor at representing hierarchical relations, and has poor quantificational power across classes.² Graph DBs have advantages for this, and it is because of the simplicity of having just two tables.

However the programming complexity now re-emerges in managing the *edge structures*. Edges are *labelled*, representing different types of relationships between points. These have their own properties and characteristics, e.g. transitive, reflexive, etc. They interact with each other – e.g. we might observe correlated patterns of *familial relations* and *hierarchical relationships* and *financial transactions* between people working in an organisation, and infer a causal relation. We really want to know how *relationships are related*. Hence the interesting level of analysis of information is at the second order. This is where the complexity of content re-emerges at the representational level, with *ad hoc*, customised procedures to handle content.

The second-order facts are not put back into the bi-graph representation with the first order facts – and there is no single domain of all facts. The representation *does not take facts as the primary entities, or direct objects of representation*. It takes a modular composition of facts *from points and edges*. The point-table alone does not represent *facts*, only entities. Edges represent facts by associating the two point-entities. But they must be from the point table. Now we also wish to represent facts about *edge types*, based on the empirical first order facts they are involved in, or on logical rules we want to impose. These are their inter-relationships, correlations, logical rules, statistical summaries, etc. But edges cannot also be treated as *first order point entities*, and we need a second-order method to quantify over them. Thus the bi-graph method breaks down at the second order – unless we reproduced the first order apparatus for the second order facts ... and then the third order, etc. But graphs databases to my knowledge are not iterated in this way.

Another important complication of bi-graphs is that each point *can have any number of in-edges and out-edges, of any types*. This means when processing functions, the code must loop over multiple edges per point, and perform processes within these loops, which in turn require loops over multiple edges per point. This kind of recursive looping seriously complicates data processing. (CAT4 graphs have exactly 4 joins to process for each point, with simple logical structures.)

² At least core SQL does, but in stronger languages, if you have to make nested iterated loops in code over class collections and properties to retrieve sets of values, quantification is still poor in our terms.

There are also problems of scaling graph dbs. The edge types can combine to form all kinds of network structures. And new edge types can be added. In a large network, the number of edges is usually much larger than the number of points. The number of *entities*, N , we talk about is relatively limited compared to the number of *binary combinations* of the entities, which is N^2 . E.g. if we have 10 people, we need: $10^2 = 100$ edges between them, to fully characterise one simple relation such as whether they like each other.

We expect the *average number of edges per point* of an existing set to increase with expansion of the size N of the point domain, since adding new points cannot remove existing edges from the existing set, and must sometimes add to them if the graphs are connected. Every relationship is a new edge type and set of edges, and temporal relations are another layer of complexity, and this proliferation of *edges* is problematic for scalability. We cannot put relationships (edges) in for all entities with relationships – every entity has *some* kind of derived relationships with every other entity. We only require *primary or atomic edges* as the primary representation. We can add logical consequences of these later.

- Suppose every point in a graph has an average of four edges. The total number of edges is: $E = 4N$, with N the number of points. This scales w.r.t. N by a constant: $E/N = 4N/N = 4$. This is scalable: it takes a fixed number of edges on average to increase N by 1. But suppose the average number of edges increases with N , e.g. by: $E=4+N/a$, where a is a constant. Then the total number of edges is: $4N + N^2/a$. If the N/a term gets too large, this becomes unscalable. E.g. the average number of edges per point doubles to 8 when: $N = 4a$. Equally, it becomes increasingly difficult to calculate logical consequences in the network graph as it gets larger.

CAT4 DB.

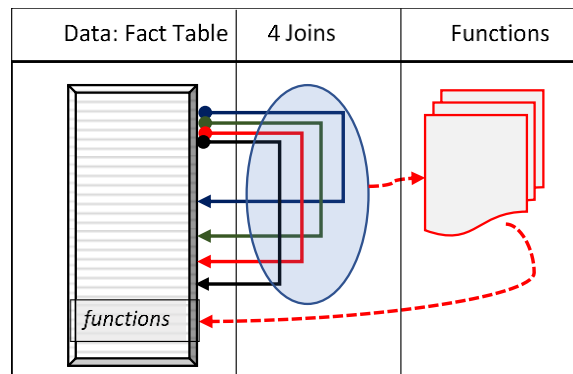


Figure 3.1.24. CAT4 representation. There is just one table. Functions and joins are defined by code objects outside the fact table, but all information about functional dependencies among points is provided within the table (4th join), and all information about the function code is provided within the table.

It is the highly structured nature of the CAT4 network that makes the functions programmable.

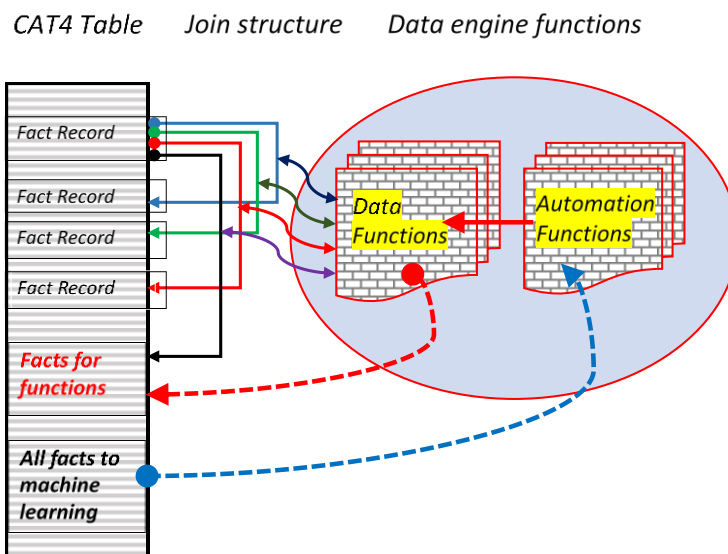


Figure 3.1.25. The simple functional nature of the CAT4 system is because facts are not typed: they all go into the same table. Functions operate on the CAT4 table. Functions are represented as facts in the CAT4 table. CAT4 records provide data for automation functions, a general base for *machine learning*. Algorithms copying previous patterns are standard.

Table networks in RDBs can have all kinds of join structures, there are few rules, and in practise we find all kinds of ad hoc structures. Similarly, graph databases can have all kind of structures, with few constraints. We end up with complex network structures that are exponentially difficult to manage. But CAT2 is the opposite: the formal structure is very simple. Given that we can transform data into CAT2, this gives a big advantage in writing algorithms for functional recursion and automation.

Scalability of complexity.

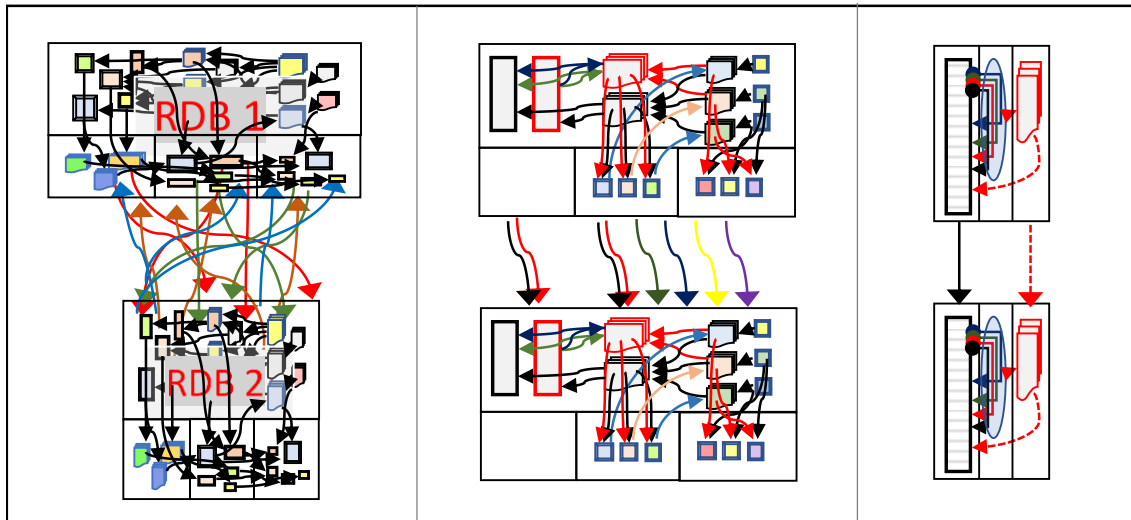


Figure 3.1.26. Left, *RDB* data exchange. Center, *Graph DB* data exchange. Right, *CAT4* data exchange. A system at top exchanges data with a system below. The exchange requires ETL procedures, arrows in the center. These are normally programmed in SQL and various code.

- When RDB's have to exchange data with other RDB's, the data has to be carefully extracted and transformed, and prepared in the right form to load into the new tables, table by table and column by column, with data rules, etc. This is shown as a big mess: it is all manual, ad hoc programming.
- Graph DB's can exchange data with each other using much more systematic procedures. The point and edge tables can be mapped to each other. However edge type data and *relationship rules* and *functions* are built manually, and still must be translated manually.
- CAT4 DBs can have a fully automated data exchange. Transactions of information are between two identical CAT4 tables. This also applies to functions, which are recorded in CAT4 data. In practise there will also be exchange of compiled functions.

RDB's are rigid, with fixed tables and naming conventions, etc. Two RDB's on the same or related subjects will usually have quite different naming conventions, and different methods for organising the data in tables. Automated data exchange (ETL) systems to map data between databases are highly engineered, and very labour intensive to build and manage. It is generally impractical to have more than a few independent RBD's in connection with each other. *Database connectivity* refers to ability to import whole tables across databases, but *integrating the data* from one RDB system into another is far more difficult, and the method is unscalable.

Graph DBs do better in some ways, but CAT4 is a far more elegant solution. Data exchange requires matching *fact* records across two tables. This started using an *index* to match known records, and is then continued by matching common parent records and 3rd joins, along with syntactic matches on

content. Existing matches are automatically recorded in the data, and build up a machine learning base. The importation process is simple once the matches have been decided. Existing records are not added; new records are added exactly as in the source CAT4. This is because all CAT4's can be combined with each other.

There is something else essential, the ability to partition CAT4 DBs into separate CAT4's containing all the information directly about different subjects. This is the: *InEx(p)* function. These are like filters. A single CAT4 table will get very large of course, and we need to partition it into smaller pieces. We can break it into partitions, and store it in these smaller tables. We can work in these partitions independently, and integrate them back together subsequently. This is impossible in an RDB, because it is made of very interdependent *mechanical parts*, and you cannot sensibly remove parts of the database or parts of the data, and set it up as an independent RDB. CAT4 has an *organic construction*, and can be split into parts, and recombined.

Scalability of volume.

In terms of data volume scalability, an initial reaction to CAT4 may be: *if we represent every fact separately, won't we be overwhelmed with the number of facts!?* And then, what about all the subsequent relationships between facts? Don't these multiply? To deal with the second point first, *all relational facts (at all logical levels) are included as facts in CAT4.* The collection of facts is limited to this. There is no realm of additional facts outside it. Hence, the total data representation has a total size proportional to the number of facts in the fact table. Of course we can add a lot of facts; but we know exactly how many we have.

Secondly, we want to represent every *significant atomic fact* we have, from a database say, individually in CAT4. Each of these *needs* to be represented, being part of the real information. (CAT4 can however leave behind a whole lot of meta-data or systems data and code required to make the DB work, as well as normalising complex data better.)

Imported data tables generally take more space initially in CAT4 than as tables, because the database uses techniques to specifically optimize table data. However, the difference is not that great, and CAT4 becomes more economic as we aggregate a large database, because it has better *normalisation*. This is helped by *semantic bundling* in RDB tables. Note that in conventional database tables, *columns* are not generally independent. Typically we associate two columns carrying a *value* and a *date-time*, or a *value* and *start and end time*. Rows recording changing values over time are generally indexed by secondary keys, (*ID1* and *ID2*), to rows in primary tables representing

permanent entities. This is a standard technique for representing *changing facts*. A table with columns representing temporal facts typically looks like this.

Keyed Fields			Account Details			
ID	ID1 – Clients	ID2 – Accounts	Value	Truth Value	Time 1	Time 2
0	Mrs. A	Saving Balance	\$20		1/1/20	1/2/20
1	Mrs. A	Saving Balance	\$12		1/2/20	1/3/20
Etc						

Figure 3.1.27. Semantic bundle. This is a typical kind of *column group* in a conventional table representing changing facts, in this case about Mrs. A’s *Saving Balance*. It requires almost identical data columns and rows as the CAT2 table to represent the equivalent facts. These would be *bundled* into a single CAT2 record.

Conventional database table columns typically form “bundles” like this because they need to make complete *propositions*, which require *times*, *values*, and *external referents*.

CAT2 typically “bundles” the data from a row of several columns like this into one *point token*. This is one reason the method becomes more economical for space once we have to store complex data. Although it initially ‘explodes’ the table data into greater detail, we usually find that several data elements in conventional tables go into the fields of one CAT2 record. This has another advantage: CAT2 associates all the values logically required to form propositions. In a relational table, we equally have to know which columns to associate with each other, but this is additional semantic information required to interpret the table, not represented in the RDB data.

In any case, every significant fact represented in the database requires at least one individual symbol, with a minimal storage depending on its *symbol type*, e.g. it typically requires *8 bytes* for a number. In the worst case, the storage for this as a fact in CAT4 is about 24 bytes. Hence the data volume might be 3 times. But this is an upper limit. The data volume in CAT4, with some simple optimisation techniques, should scale more or less linearly with the data volume of a simple RDB, but will become more efficient with more complex databases.

The CAT4 network is extremely efficient at representing complex relationships and functions. The *logical and semantic relationships* are given by the join structure, with the fixed number of joins per fact. It is only *tips*, typically cell-values imported from large RDB tables, that are any cause of concern for storage or volume. But there are several ways to deal with this. Even if data took three times the space, the increase in computational simplicity and power far outweighs this cost.

3.2 CAT3 Semantics: reference.

Reference join concept.

We formally introduced the 3rd join in Part 1.

- Each point has a point-value for ID_3 , extending CAT2 to CAT3.
- We will sometimes write this with a special symbol: $i \rightarrow l$. This is a function.
- We can define C3 relations with 4-tuples of points. If: $(i,j,k,l) \in c_3$, then l is the 3rd join or 3rd parent of i in c_3 .

The axioms for the 3rd join are discussed below, and are intended to capture the semantic concept of *two points (fact tokens) having the same object reference*. This is a primary semantic concept, and we will start to consider CAT3 as a *graphic language*, rather than just a ‘data storage system’.

But to start with an image of the 3rd join, suppose we have a relation, as above, called *Clients*, with *Mrs. A* as a member (individual), with some properties. And we also have a second relation, called *Authors*, also with *Mrs. A* as a member. Two distinct points refer to the same entity, i.e. *Mrs. A*. This information *must be formally represented* if the database is to be semantically complete. It can be represented most directly with a 3rd join between the two points.

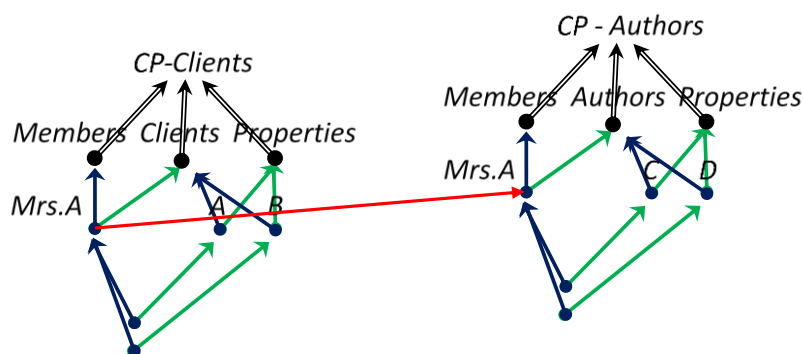


Figure 3.2.1. These are two sub-graphs of a single CAT3. The two points labelled “Mrs.A” both have the same reference, the person, *Mrs. A*. The red arrow represents a 3rd join.

The 3rd join represents *object identity* independent of *names*, e.g. *Mrs. A* may be a doctor and publish books as “*Dr. A.*”, or under a pseudonym, but we would still join the points as long as the two names refer to the same entity. This 3rd join represents essential information, e.g. telling us that a person with certain Client attributes also has certain Author attributes in a second relation. Indeed, 3rd joins are a major part of the representation, and must be *completed fully and accurately* for the CAT3 graph information to be entirely correct.

Note the two point-tokens for “Mrs. A” are used to state two distinct *facts* – first that *Mrs. A* is a member of Clients, second that *Mrs. A* is a member of Authors. The 3rd join tells us that there are not *two individual Mrs. A’s*, there is *one individual Mrs. A*, involved in both facts. The two equivalent point-tokens are used to represent two distinct facts about the same referent.

We can see points in a CAT3 graph as being like the words of a language, combining with other points to represent complex facts. CAT3 graphs (relations) are the complete, self-contained *expressions* of the CAT3 language, analogous to *sentences* in natural language (NL). The ‘same words’, i.e. fact tokens labelled with the same referential symbols, may be used repeatedly in CAT3 expressions, just as the ‘same words’ are in NL sentences, to make the same references³. This is formalised by the 3rd join. This is the intuitive concept we want to capture.

- Two fact tokens with the same reference are always related in a tree of 3rd joins.
- Fact tokens in the same 3rd join trees form *equivalence classes*, represented by primary point (trunk) of the tree.

Reference is a property of every point. Every point has a unique referent, which may be null. Points may be taken to refer to external entities of all imaginable kinds, individuals, objects, properties, property values, relationships, numbers, functions, and categories or classes of all these. For those concerned about reference to abstract objects or fictional entities, we characterise this within the data. We do not need to interpret what the referents are, or decide whether they exist, to start recording facts. CAT3 is a formal calculus, and we only need assume that we can identify terms that have the same referential meaning within CAT3.⁴

This semantic relation must be specified formally, independently of the symbols (names) we use for fact tokens, because we use many names for the same things and the same name for different things. The *syntax* of natural language only gives clues to the semantic interpretation: we want to represent propositions in a fully functional or compositional system, which requires a precise formalism for the *semantics*.

³ A key difference with NL is that a single CAT3 can encompass all facts known by an agent at once, and provides a complete and transparent logical relation underpinning facts; while NL communication represents facts in fragmentary sentences, representing propositions about parts of the world at a time, without transparent logical relations between sentences. But the underlying concept of *construction of complex representations* using a base of *common referential terms* is the same.

We have been assuming previously, in reading our diagrams, that we can recognise *referents of fact tokens* from their *symbolic content*, or labels. We do this in practice of course by their association with words in natural language. We cannot identify the *external referent* of a point from within CAT2, so speak. But we can take *referential equivalence of symbolic tokens* as a fundamental semantic relation, and represent this within CAT3.

Thus we need an equivalence relation between *all fact tokens (words) with the same referents*. Note our *3rd joins* only represent referential equivalence of pairs of terms at once. But the relation is transitive, and forms equivalence classes of points, through *3rd join trees*, also called *reference trees*. We call the points at the top *primary points*.

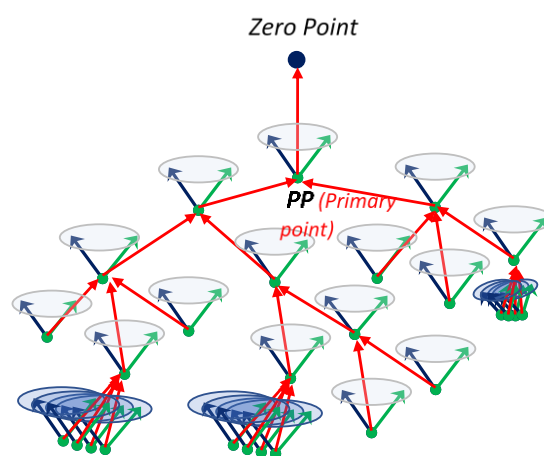


Figure 3.2.2. The *3rd join* form *trees*, going up to a *primary point*, which then goes to the *zero point*. The *primary point* (PP) is the representative of the referential equivalence class. Each point exists within its local *CAT2 lattice*, *3rd joins* can go across lattices, or within CP's.

The *3rd join* structure we have adopted is more complex than needed just for equivalence classes. We could have every *3rd join* going to a pre-defined primary point (as in formal ontologies). But there are several reasons for having trees of *3rd joins*, instead of simply joining every point directly to a *primary point*. First it takes no more data. Second it is very convenient for representing *hierarchical sub-classes*. Third the trees represent contextual relations for the use of terms, and give similarity relations among them. Fourth, trees grow and are completed as we improve our information, and

⁴ Note points can also refer to entities like functions or procedures, which may also include computer-defined procedures that we can execute. However, we *implement functions* separately to *reference joins*, as they are dynamic procedures executed over the CAT4 relation. Also, we do not generally use reference joins for *equivalence of numeric values*. Numerical equivalence is syntactically processed as usual, using number formats. See later.

we often discover and record identities between pairs of entities directly, without working through a third logical entity, a primary point. Fifth it is easier to edit information about identities by joining and switching branches of trees, rather than switching joins to a new PP *en mass*. These *identity trees* therefore also contain information beyond simple referential identity: they help *categorise identity inferences*, and this is very useful to the dynamic functioning of the system.

Primary points and primary lists.

The fundamental reason we need this relation is because we have to represent facts about *the same entities* using different tokens in different places in a CAT3 graph. E.g. in the previous graph example, we represented facts about *Mrs. A as a client*, *Mrs. A as an author*, *Mrs. A as a relative*, etc. These are facts about the same person, *Mrs. A*, but they cannot all be represented around a single point ('word') representing *Mrs. A*, because they are logically independent, and have different parents, etc.

In this example, we have taken *Mrs. A, the Author* as the primary occurrence, and *Mrs. A, the Client* as a secondary occurrence. But this order appears arbitrary (unless perhaps all *clients* come from the *Authors* table.) Note the common way to deal with this in a relational database is to have a primary table of *all persons*, and use this (via SQL outer joins) to relate identities of all persons appearing in various other tables. This technique is useful in RDBs, but very limited. We illustrate this in CAT3 next. But note that CAT3 gives a systematic way to assign reference for every fact.

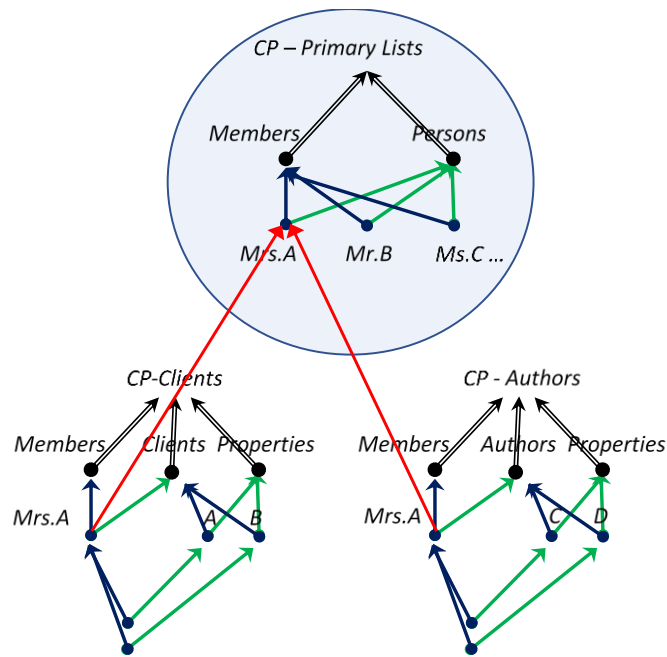


Figure 3.2.3. Here points representing “Mrs. A” occur in two distinct places, and both are joined to “Mrs. A” in a *primary list of all persons*.

Primary points act like an *ontology*. Referential identity applies to *all facts and all concepts*, even very abstract ones. E.g. the “membership” relation.

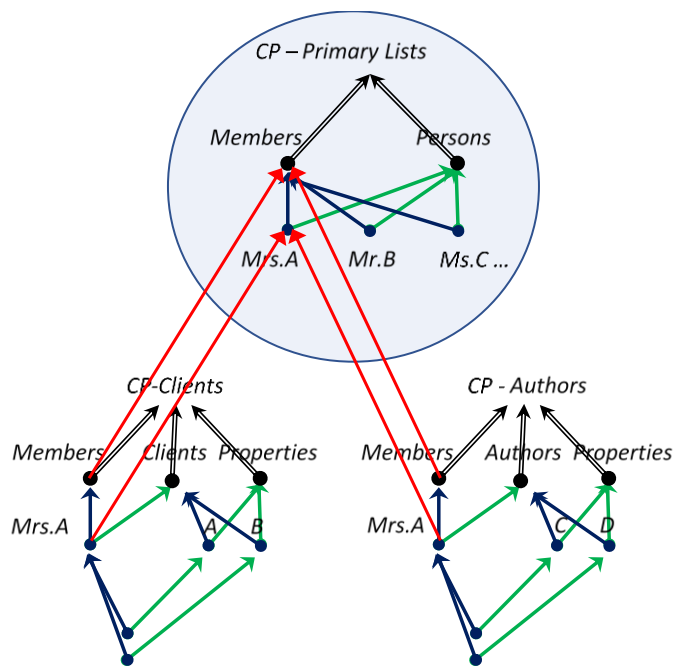


Figure 3.2.3. Now we have added 3rd joins for the ‘Member’ relationships.

Here we identify the *Members* relation in the Authors and Clients graphs with the *Members* relation in the Primary Lists graph. It is reasonable to adopt this common interpretation of “Members”, as the three “Members” relationships are intuitively the same concept, and they correspond formally.

In each case, some of the same individuals (persons) are right children of *Members*, showing they are consistent (functional) types.

But note that in CAT3, *primary entities*, at the ends of 3rd join chains, do not have to be organised in distinct lists. They will be scattered throughout different parts of the graph. CAT3 networks can have a variety of 3rd join arrangements between points, and they can change as the network evolves. They can be *normalised*, and organised in logical patterns, e.g. by collecting all primary referents from a category in a single list, and re-ordering reference chains. Note such joins are maintained continuously when data is inserted or deleted, and there is only *one 3rd join per fact*, so the representation is scalable with facts.

We sometime use the notation: $A \rightarrow B$ to mean that *point A has its 3rd join going to point B*. Note *A* and *B* are two distinct fact tokens (points in the graph), and these state different facts, through their different places in the network. But as purely referential symbols they *refer to the same thing*, like repeated words in sentences.

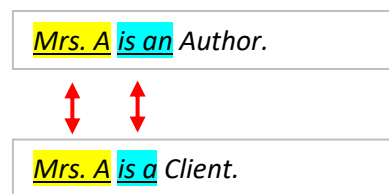


Figure 3.2.4. Two sentences with words referring to two identical concepts.

Note the difference in syntactic form between: "is a" and "is an" is just an idiom of English grammar, telling us that one is followed by a consonant, and the other by a vowel. In terms of conveying propositional information, they both signify an identical relationship, between *Mrs. A* and the property of being an *Author* or *Client* respectively. We have identified this as the *Members* relation in the CAT3 graph above. When we read a CAT3 diagram, to make sense of the 3rd join, we can ask:

- Can we replace the referent of the fact token with the referent of its 3rd join?
- Can we state *the same properties* of the 3rd join referent as we state of the point referent?

The comparison of reference term-by-term in the two propositional statements, above, with the previous diagram, confirms the third join used there makes sense in this way.

We next review the formal properties for the 3rd join, which are designed to be adequate to represent this semantic relation.

Third join formal structure.

We specified the formal properties of the 3rd join in Part 1, Axiom 5.

(Axiom 5) If: $c_2 = \{(i,j,k)\}$ is a C2 relation, an extended C2 relation with point-relations: $c_3 = \{(i,j,k,l)\}$ is C3 just in case: (a) The relation: $T_3 = \{(i,l): (i,j,k,l) \in c_2\}$ is a tree* (directed rooted in-tree with $0 \equiv (0,0,0,0)$ as the root), and (b) c_3 has no cycles and (c) the point l is not in the (C2) interior of i .

Intuitively, the first part (a) and (b) means that every point has a 3rd join to another point, creating *reference chains, with no cycles*, so 3rd joins form *trees*. The zero point is the root* (self-joining*). Points (“trunks”) joining to the zero point are the *primary points*. They are collection points for all facts with the same *referents*. The class of *primary points* represents an *ontology*: a set of facts corresponding 1-1 with entities referred to ‘in the world’. The 3rd join thus induces an *equivalence relation on the class of facts*, with each *primary point* as the representative point, and its *exterior* as the equivalence class.

The reason for preventing *cycles* is to prevent self-referring entities or functions. We cannot define *reference* in a circular loop. We cannot define *objects* as being parts of themselves. Nor can we define *facts* as parts of themselves. This gives self-reference paradoxes, like Russell’s paradox.⁵

This need to avoid circularity excludes the 3rd join of a point joining into its own interior. Otherwise the interior point represents *a fact containing a fact about itself*. The referent of a fact would appear among its own values (in the exterior), and we would get an infinite regress in the exterior. This is why we have (c) in the 5th Axiom.

Note that we also cannot allow the 3rd join of a point to join into its own exterior, but this is ruled out already, because it would create an ordinary cycle in the three joins. I.e. you could go up from p to q (by left and right joins), then return to p by the third join: $q \rightarrow p$. The special rule for the third join *prevents direct interior-exterior relations between two facts that have the same referents*. An abstract example of a 3rd join tree structure is illustrated.

⁵ The simplest example is the statement: “This statement is false”. Paradoxical statements of this kind are not simply false, rather, *if we take them to state propositions, they are both true and false*, and logically incomprehensible. We want to rule them out because they do not really construct any proposition, and if we allow them in a formal system, they threaten to destroy the logical framework. In CAT4, paradoxical interpretations are prevented by rules preventing circular dependencies or cycles in the network. This means paradoxical statements cannot be interpreted with their apparent paradoxical meaning – it is not representable as a proposition or fact.

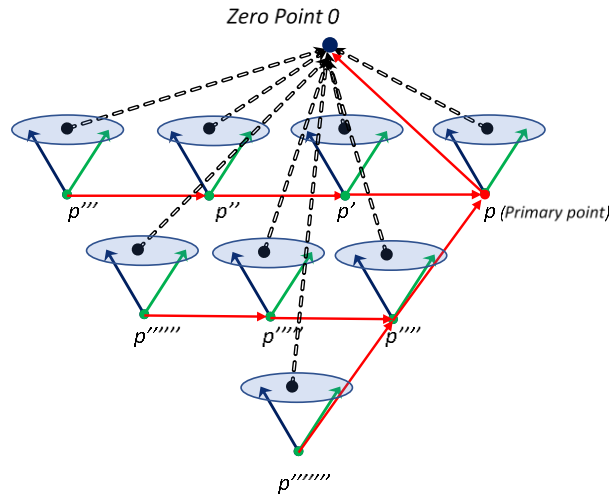


Figure 3.2.5. This illustrates a random 3rd join network, which forms a tree looking down, (red joins), with the zero point as the root, and the primary point p as the trunk, and p' , p'' , etc, in branches of different lengths. Each point belongs to a local CAT2 lattice, with a collection point (CP).

This diagram indicates that the 3rd join tree (of p 's) is separated from the CAT2 structure (blue and green joins), but this is simplified, and facts in a reference chain can intersect in their interiors, as in following examples. The rules for inserting a 3rd join are most simply as follows, for all points except the zero point:

- The 3rd join cannot go to a point in the (1st & 2nd join) interior or exterior.
- The 3rd join cannot go to a point in the 3rd join exterior (or it creates a cycle).

The 3rd joins which we symbolise: $A \rightarrow B$ may be directly from a secondary point A to a primary point B , but they can form chains, e.g. $A \rightarrow B \rightarrow C$, where A is first identified with B and this is then identified with C . At the end of every chain excluding 0 we have a primary point, and this has its 3rd join set equal to 0 , referring to the zero point. (Note: later this is changed to the CP point, giving intensional context via the reference chain).

In one sense this “0” value could be taken as “none” or “no further referent”. Every point for which we cannot specify a prior existing reference within the graph has its 3rd join set to 0 , as there is no further point to pass the reference on to. But semantically, we can interpret the zero point, 0 , as referring to “the world”, the source of empirical facts or empirical truth. So for primary points, we take the 3rd join to refer us to the entities in the world, and this means we must leave the graph and look in the empirical world for the referent. We cannot understand what the information in the graph represents empirically unless we have a connection to the real-world entities.

CAT3 file-folder hierarchy representations.

The next example shows the use of the 3rd join to represent file-folder hierarchies. File relations are tree hierarchies, with *folders as nodes*, which may contain both files and folders, in trees below. To represent this in CAT3, we first represent a collection of *all files and folders*, illustrated below as *Members^v Files*. In this example we illustrate with only one folder (*A*) and one file (*B*), but many files and folders can be represented in the same positions. *Folders* are regarded as *collections of Files (including other Folders)*, and we need to represent the ‘membership’ relation for such collections. To do this we must relate three entities at a time: *a folder, a file, and the folder-file relationship*. The obvious way would be to join them as below. However, CAT2 does *not* allow this!

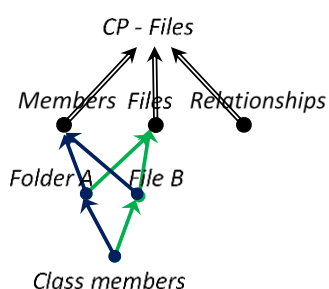


Figure 3.2.6. CAT2 does *not* allow this “Class members” point, joining to two points with the same parents. Two points must be *adjacent* to be able to be joined by a third point below.

Instead we can represent membership as below, with a second “image” of *File B* in the appropriate position to represent the relationship with *Folder A*.

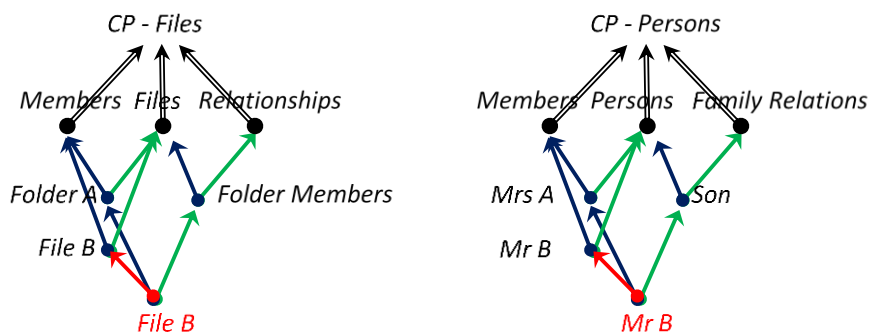


Figure 3.2.7. Left illustrates a method for representing file-folder memberships, and represents the fact that *File B* is a member of *Folder A*. The points in red are ‘image points’ of the primary points they refer back to.

Note the right graph illustrates a way of representing family relations among persons, representing: *Mr. B is a Son of Mrs. A*. This appear formally identical to the first graph structure.

We know the 'image point' "*File B*" represents *File B* because of the 3rd join. We can do this with any number of files-folders, and represent any folders or files as class members of other folders or files.

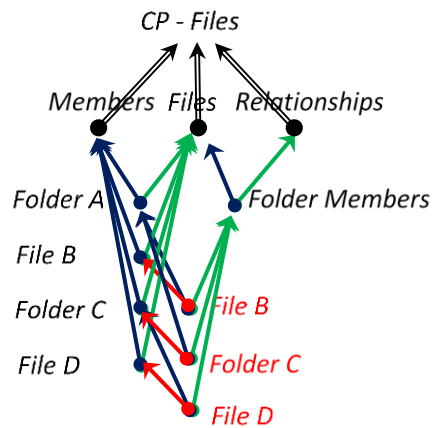


Figure 3.2.8. File B and Folder C are both in Folder A, File D is in Folder C.

Although it represents a folder hierarchy that may be many levels deep, the representation is only two levels deep in CAT4. We see the folder-file relation, normally pictured as a tree, as an hierarchical binary relation. However, this leaves us with an implicit bi-graph type structure to deal with, and is not the most general method.

CAT3 binary relation representation.

There is a second way to represent such relations, which is more in keeping with the concept of binary relations over single domains of entities, as with the *Son* relationship in the second example. This method is to make a *second copy* of all the entities (e.g. *files* or *persons*) in an adjacent position to the *primary points*, so we can make direct relationship joins like this.

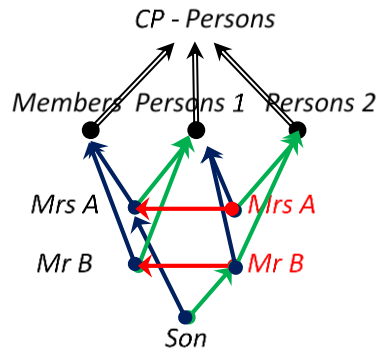


Figure 3.2.9. We copy elements from the list of *Persons 1* into a second adjacent list, *Persons 2*, and identify them with 3rd joins. This now allows us to join them (below) as *ordered pairs*. We add points representing relationships. This allows us to insert relationships on the *cross-product*: $\{Persons\} \times \{Persons\}$. But we still need to identify the *Son* relationship.

However, this is not fully satisfactory, because we have labelled the relationship between Mrs. A and Mr. B as “Son”, but we will need a 3rd join system to identify it as *the same relation* that holds between other persons. E.g. if Mrs. A has a second son, say Mr D, then we end up with two points both representing the “Son” relation, and we need 3rd joins formally representing this. There are two ways to deal with this.

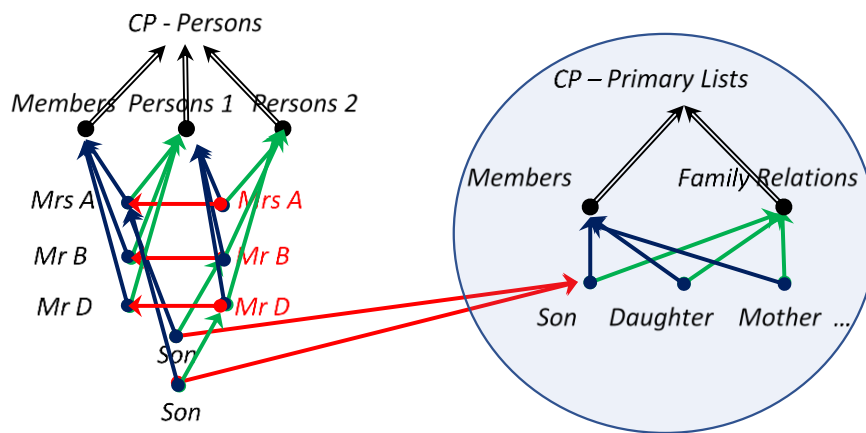


Figure 3.2.10. We can identify that the two relationships labelled “Son” in the graph are the instances of same relationship, using 3rd joins. These are normally directed to a “Primary List”, cataloguing all entities, relationships, properties, etc, across the graph.

There is a more general method again. This is to represent *ordered pairs of entities* (*Persons* in this case) in this cross-product of $Persons 1 \times Persons 2$, and then to add points for binary relationships below these. We see this next.

CAT3 Cartesian Product representation.

For a generalisation, we now represent general relations on 2 -tuples of entities. We take *relation-instances* as *values of ordered entity-entity pairs*, rather than as above, where *entities* are *values of entity-relation pairs*. This then allows us to represent symmetric relations, equivalence relations, or any relations over finite sets of entities, which are the domains of the relations.

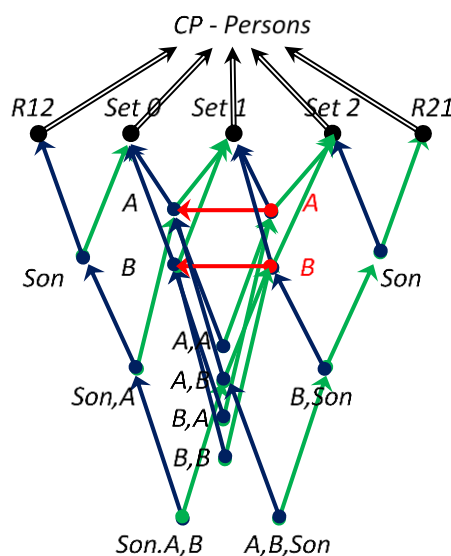


Figure 3.2.11. Representing relations on cross-products of set domains. We can insert ‘relationship facts’ on the left or right of couples like (A,B) . This technique can be applied to arbitrary sets, to give N -dimensional cross-products. This also illustrates the lateral symmetry of CAT2 networks. Note the product does not need to be saturated; and there can be multiple values per position.

In this representation, we will take the two *propositions* represented by the facts at the bottom to be identical. Note the converse proposition, that: *Mrs. A is the Son of Mr. B* must be made by joining *Son* to the reversed: (B,A) couple instead. With asymmetric relations, the meaning of the order of arguments is conventional, but it must be consistent across a system of relations.

There are only two 3^{rd} joins shown. Note the change of structure from previous graphs, which use more 3^{rd} joins with a narrower/shallower CAT2 lattice, to a wider/deeper CAT2 lattice with fewer 3^{rd} joins, and three horizontal categories: *person 1*, *person 2*, *R12*. This is a kind of *normalisation into a dimensional structure*. Apart from the joins on the *base sets for the cross-product*, which is like an ontology, we have lost the multiple 3^{rd} joins on the data points required in some earlier diagrams.

This makes this representation easier to query, and gives a space within the local graph CAT2 to represent the *relations* as primary entities, give them properties, etc. Graphs using the CAT2 lattice

for propositional constructions, rather than 3rd joins, can be simpler to query. With our new graph, we can now query data by choosing individuals (e.g. A or B) from Set0[∨]Set1 or Set1[∨]Set2 and relations (e.g. Son) from the relations classes: Set2[∨]R21, and taking their CAT3 Exteriors, E.g. Ext3((A,B)/Son) gives all information combining the Son relation with A or B.

When we query for information against *subjects*, which are themselves just selections of *points* (*facts*), we need to: (A) find the exterior of their 3rd join trees, then: (B) find the CAT2 exterior union/intersections of these. This process essentially gives us all the querying functions we need. We generally end up with an alternating chain of queries: (3rd-join tree → exterior unions/intersections → 3rd-join tree of results → exterior unions/intersections). There are efficient scalable general processes to calculate the EXT, INT and INTEXT functions on the CAT2 lattice. And similarly, calculating the 3rd-join trees can be done efficiently. However, the more 3rd-join trees and stages in the chain, the longer the query takes.

CAT3 examples of non-relations.

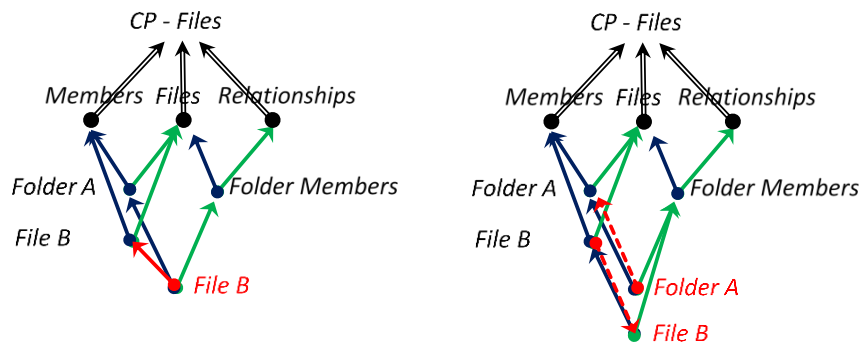


Figure 3.2.12. Left graph is permitted in CAT3. The interior of the “File B” image point has “Folder A” and “Folder Members”. The third join goes to “File B”. This is not in the interior or exterior of its image point, or in its 3rd join tree. Right graph is not permitted. Folder A image point cannot join to its interior and File B image point cannot join to its exterior.

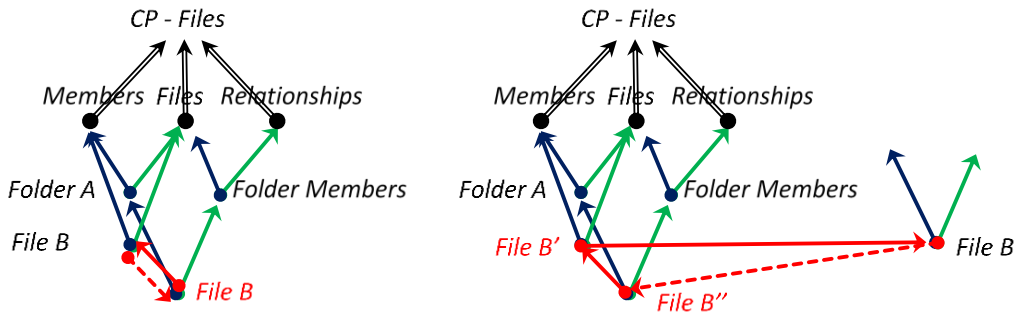


Figure 3.2.13. Both illustrate 3rd joins that are not permitted because they form cycles. Left would form a 2-point cycle, right would form a 3-point cycle. This is equivalent to the condition that points do not join their 3rd join exterior tree.

“Circular relationships” can be represented within this structure, however.

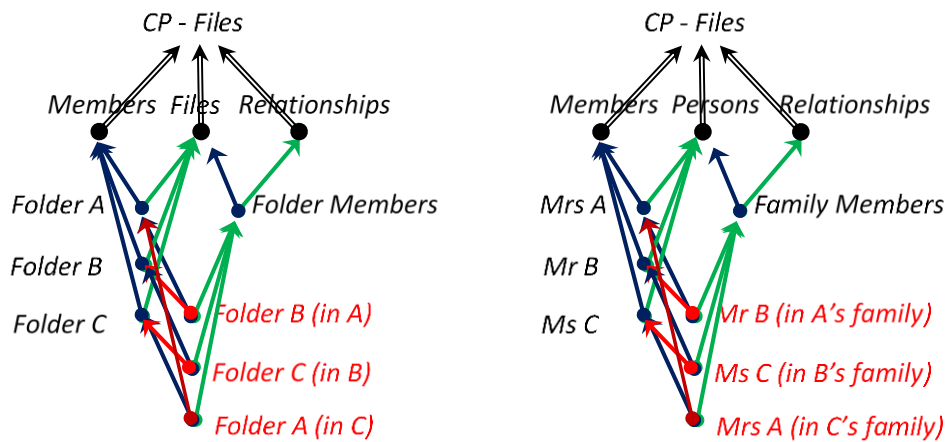


Figure 3.2.14. CAT3 permits *circular relationships* like this. Left represents *Folder B in Folder C, Folder C in Folder A, and Folder A in Folder C*.

This may seem like a circular paradox, but it is not *paradoxical*, it is a logically possible relation. It is just wrong for folder-file structures. The right graph represents the same *formal* relation, now interpreted as: *Mr. B is in Mrs. A's Family, Ms. C is in Mr. B's family, and Mrs. A is in Ms. C's family*. This makes sense.

With the Folder example, we would normally say that one or more of the facts represented by the 3rd joins must be *false*, because it breaks the normal membership rules for Folders. It contradicts our intuitive sense of the logic of ‘containment in collections’, or ‘class membership’: we cannot make a folder a member of another folder that contains another folder that contains the first folder. This seems logical – for folders. But this is just a particular type of relationship, applying more generally to *tree relationships*, including *folder structures*.

This method lets us instantiate binary relationships, but with only some varieties of *a/reflective*, *a/transitive*, *a/symmetric* properties. E.g. we can form relations that are *reflexive*, so if: (A,B) then (B,A) . Or *irreflexive*, so if (A,B) then not (B,A) . Similarly we can form some transitive relations, but:

- We cannot use this method to represent a *symmetric relationship*, because we cannot insert an *image point for A in the exterior of A*.

Hence we also cannot represent an equivalence relation in this manner. This method is suited to *file-folder hierarchies*, or *collection-instance-property hierarchies*. It is commonly seen when entities are naturally at different levels, of different types, or part-whole hierarchies.

The binary relation and Cartesian product representations are more general, and required to represent many common relationships.

CAT3 analogical and recursive representations.

There are two ways to represent various “geometrically-patterned” data, such as file-folder hierarchies, in CAT3: with an *analogical* or a *recursive* representation. There are variations of both.

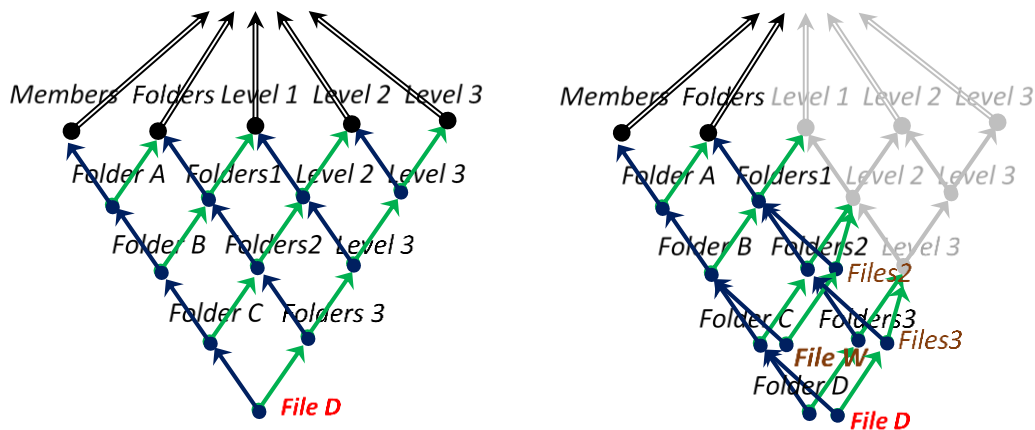
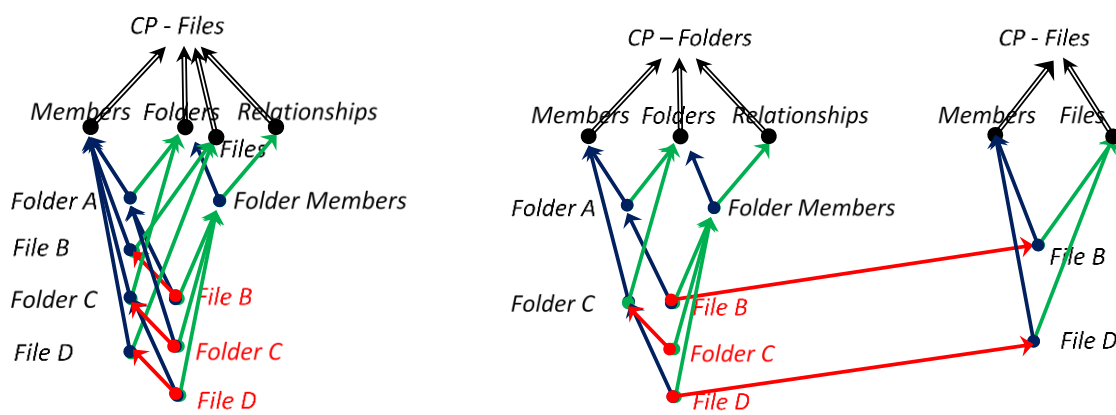


Figure 3.2.15. Two slightly different *analogical representations* of a folder-file tree hierarchy in CAT3. The spatial relation in the graph reflects the hierarchical membership relation, with folders as *collections*. Or a tree structure. Left, we store *File D* in the folder hierarchy. But this does not distinguish *folders* from *files*. Right, folders are separate collections of *folders* and *files*. The *files* are distinguished from the folders by the right joins.

The graph-method on the right contains more information than on the left, by distinguishing *files* and *folders*. But it now requires a *Folders* and *Files* parent at every *Level*. Note on the far right we have shaded the points under *Levels*. These interior points are identical for all points on the left hand side of the graph, where we add the real information, and they do not provide additional information.

This does not use 3rd joins (like the *recursive representation*). But this kind of analogical method is generally not good, because it uses too many levels. Its critical information is contained in long *interior chains* of points. It distributes points that really represent *the same type of entities* over different positions in the lattice. This is possible to represent relationships, but these should not be *primary points* for the entities. The lattice joins generally represent *logical joins*, not *analogical joins*. The analogical method here defines the *identities* of files and folders by their interrelations in the network, instead of a defining them as individuals in single classes, of the same types. Membership relations should be added as exterior points, representing contingent information. This gives a shallow structure, with only one level of quantification required over the *membership relations*.



Recursive composition of hierarchy. Left. This distinguishes *files* and *folders* through right parents, and all folder membership instances are under *Folder Members*. Because this point joins to *Folders* on the left, only points under *Folders* (and not those under *Files*) can have instances of *Folder Members*. Right. To emphasise the separation, we can separate the *files* from the *folders*, and move them to another position if we like. We only *point to files* with 3rd joins, they are not involved in the *folder hierarchy*.

In this method anything can be a member of a *folder*, and we can have 3rd joins going to various CPs across the network. This is similar to defining *folders* as general classes, without specifying a structure of *levels*, as in the analogue version.

The *recursive* method has the same basic *atom* as the *analogical* one, viz. the one-level relationship has essentially the same proposition and graphic relation.

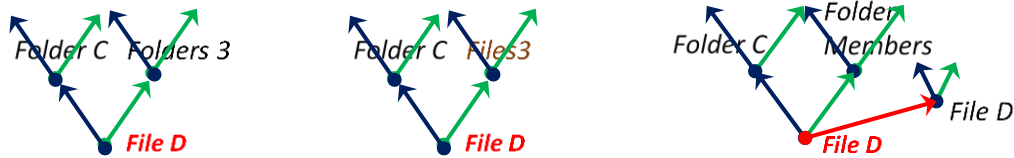


Figure 3.2.16. These atomic relations are essentially the same, but the left hand one is really the most accurate. Left, Analogue. The two versions of the *analogue* atomic proposition: *File D is in Folder C*. Right, Recursive. The *recursive* atomic proposition: *File D is in Folder C*.

The recursive relation changes the representation by using recursion over this type of atomic proposition, to build up the hierarchical relations. This means representing the many instances of the *folder membership* propositions in *the same relative position in the network*. We can quantify over all points in this position. The analogue method distributes these propositions across the network, and makes them appear as distinct types of propositions, with no single quantification class.

Note also in the recursive method, the points representing the *members of folders* do not represent *primary entities*. The analogical method *defines the entities through their relations, and thus through their positions in the folder hierarchy*. The recursive method defines the entities in classes, as *folders* or *files*, and adds information about the *member relationship* through the exterior relation.

This shows the simplicity of the recursive method. This difference between *analogical* and *recursive* methods is essentially. We have a tendency to jump to *analogical representations*. This is good for pattern-seeking. You might say that mathematics exploits recursion over these intuitive representations, when we see they have a structural repetition.

CAT3 object part-whole composition.

The theory of composition of *wholes* from *parts* is called *mereology*. This is typically visualised in terms of physical parts, although mereology is meant to be a more general theory of *part-whole composition*. We illustrate with a physical example. Some physical things are parts of others. E.g. the Solar System is a part of the Milky Way galaxy. The Sun and the Earth are parts of the Solar System. It may seem intuitive to us to picture this as a hierarchy of classes, or collections, similar to the tree of folders (collections) containing either more folders (more collections) or files (entities). But the tree of entities now goes down in physical scale, with “physical containment” as the basis for part-whole relation. We also think of machines like this, as having components that have components.

But there is a severe problem with this: this hierarchy we imagine does not really exist. We are not really imagining any single hierarchy of entities: we imagine any variety of *conceptual levels*, and different part-whole relations, ambiguously. What we want to emphasise here is that it is not a *logical hierarchy of entities*. The basic problem is that all these ‘physical parts’, although they are on different scales, are themselves logically alike as individuals. They belong to the same general type, the category of “physical systems”. They have the same *types of properties as each other*. E.g. we can compare the volume, mass, energy, luminosity, etc, of the Earth, Sun, Solar System, Galaxy, and equally with contents of test tubes, organs, cells, molecules, atoms. These are not constructions of entities of different logical types. Physical scale does not reflect logical type. Small objects are just as much objects as big objects in logic. So *physical parts based on physical containment is a physical (contingent) relationship, not a logical one*.

People also discovered the existence of these things independently, e.g. we knew about Earth before the Solar System, and about the Solar System before the Galaxy. The logical identity of one cannot be dependent on the logical identity of another, even though one object turns out to be a *physical part of another*. E.g. we may say that *the Milky Way contains our Solar System, and our Sun, and our Earth*. However *our Solar System also contains our Sun and Earth*. Does the Milky Way contain the Earth twice, once as Earth alone, and again as part of the Solar System?

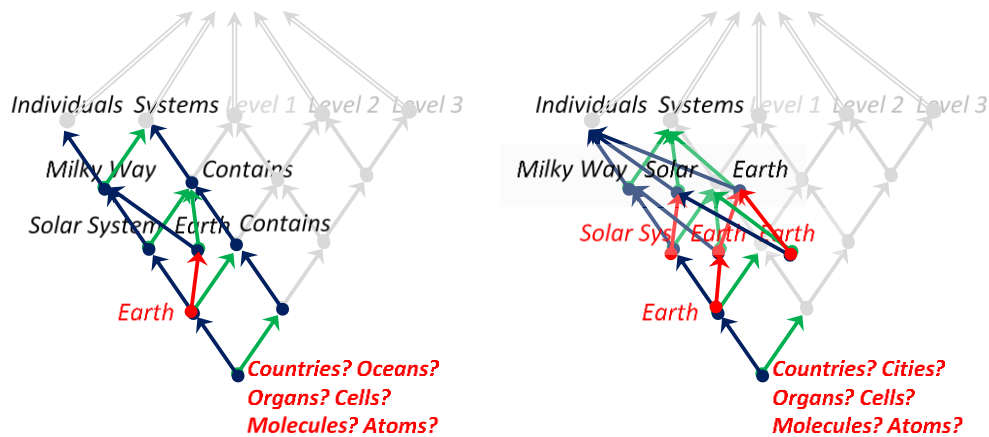


Figure 3.2.17. Analogical and recursive, combined. Left. If we represent Earth as both a member of the Milky Way and of the Solar System, a second fact about Earth is needed. We must assign the referent of the lower point as the higher point, to retain a single primary referent. Right. The Solar System must be in the same category as the Milky Way, and therefore as the Earth too. All “physical systems” must be able to be represented as individuals in this category. Exterior joins now represent a nested hierarchy. Earth is now represented as a member of the Milky Way, and of the Solar System twice.

There are also many alternative decompositions that could be given, at every level. (How should we continue to decompose the “Earth” into parts?) These decompositions reflect convenient descriptions, and follow our language, and scientific analysis. Our language reflects contingent knowledge: we have words for “natural kinds” of things, that appear repeatedly in our environment. Science has been exploring multiple layers of structures that appear at different scales, from the microscopic to the galactic, and extending our language. But there is no complete language for all natural kinds, not across all different scales, nor within a single scale, such as that of living organisms. No single objective class of *physical scales* exists. No complete set of part-whole relations between individuals or types can be represented. In our language, and databases, we represent *significant facts*.

E.g. significant facts may be given as stock-takes of the material objects inside the Solar System, for instance: planets, moons, asteroids, comets, dust, alpha radiation, EM fields, stars, black holes, Or at the level of cells: Cell wall, mitochondria, nucleus, fluid, ... Empirical knowledge in these subjects is gained by studying the contents of systems, conceptualised in terms of what appear to be their ‘natural parts’ or ‘functional parts’. We could give these lists in other terms, e.g. the numbers of atoms of different types, and this would be significant for some people, but the conventional lists of planets, etc, are the primary *significant facts* for most of us to understand what the solar system is made from.

This information should be represented in the *exteriors of points*, as shown above. This means we use the *recursive method* to represent the *part-whole relation*, in just two levels. We may extend exterior network below to represent specific hierarchies analogically. But these lower points must refer back to higher points.

It is evident that a tree hierarchy cannot represent the *object containment relation* for several reasons.

- **Mathematically.** *Physical containment* is a (vastly) more complex topological structure than a ‘scale hierarchy’, and the way we define levels of scale in such hierarchies is only a *contingent classification*. E.g. it is contingent that solar systems like ours are in galaxies, as it is contingent that galaxies formed.
- **Semantically.** If we represent the Solar System with the Milky Way in its CAT2 interior, so it is logically part of the Milky Way, we cannot represent its concept except by referring to this galaxy. But scientists knew we were in the Solar System by 1800, but they did not know we were in a *galaxy* until the C20th. If we *define the Solar System* as being in the Milky Way

galaxy, this would not make sense to people in 1800. Yet they understood perfectly well facts about the solar system, how many planets, etc.

In terms of the theory of mereology itself, a *part-whole* relation is presumably a relationship that has certain *properties*, e.g. *later than* or *distance* or *fatherhood* is not a part-whole relation. They have the wrong formal properties. So what are the conditions or axioms for part-whole relations? Well since CAT4 is the part-whole relations among *facts*, and all part-whole relations can be represented in this way, we would suggest CAT4 itself is the theory of mereology.

CAT3 class-member relations and negative predicates.

We now return to the example of *Membership* relations we saw earlier in the *Clients* example. Note there may be other class-like relationships that are distinct from this “Membership”, e.g. we might add a relationship called *Non-Members* to explicitly record individuals who are *not members*. Normally in mathematics we regard “membership” as a *logical class relation*, but in CAT3 representations of classes, it is an empirical relation. Being a *Member of Clients* is an empirical attribute of persons. But *Clients* is not simply a *class*, and *Membership of Clients* is not a class relation. Rather: $Members | Clients = \{a,b,c,\dots\}$ defines a class, which we normally call “clients”. However, we might define: $Non-Members | Clients = \{d,e,f,\dots\}$ as another class, containing persons who are *Not Members of Clients*. Being a *Non-Member* is still a *relation with Clients*, just as much as being a *Member*. There are many other classes defined for *Clients*, e.g. $Clients | Properties = \{Age, Balance, \dots\}$ represents a class of *Properties of Clients*. Indeed, there is a general function: $Clients(p) = Clients \vee p$, generating a class from *Clients* for any point p .

Hence the relationships we label “*Members*” or “*Members of*” in our graphs are not *logical class membership relations*, they are empirical relations. An important illustration is the treatment of negation. The *Non-Member relation* is an example of a *negative predicate*. We can define negative attributes as the *negations of positive attributes*. E.g. to be *not-red* is *not to be red*. If A is an predicate, we may define the negative predicate:

- Definition. \tilde{A} is the negative predicate of A if for all p : $\tilde{A}(p) = \text{True}$ just in case $A(p) = \text{False}$.

But in RDBs, negation of facts normally depends on *the absence of positive facts in tables*. In RDBs, we typically assume that *table records* are complete, so e.g. a person is *not a member of a class* just in case they are *not recorded as members* in a table. This is why there is a need to keep business databases *complete* – so we can infer negative facts from the absence of positive facts. (They *did not pay their bill* because there is *no record of them paying their bill*.) The assumption that a RDB table is

complete allows us to apply the *law of the excluded middle* for the table. But *completeness* is a special assumption, not true for information collections generally. In most situations in real life we work with incomplete sets of information.

E.g. we might record all the clients of a particular company, but we are never going to have a complete record of all the *authors* in the world. But we may establish that a certain person is *not an author*, or *not a client*, and wish to record this information explicitly.

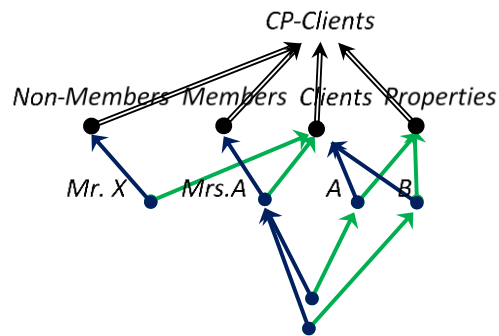


Figure 3.2.18. Clients table-graph, with facts explicitly added about which individuals are *not members of clients*. This raises questions about how to represent the negation of facts.

This example illustrates that there may be properties or relationships of *Clients* other than the *Membership* relationship. And these may be represented on the same side as the *membership* relationship in CAT3. The use of a *negative predicate* to represent *negation of a positive predicate*, is a special example that brings us to the role of truth value and propositional logic.

CAT3 propositional equivalence.

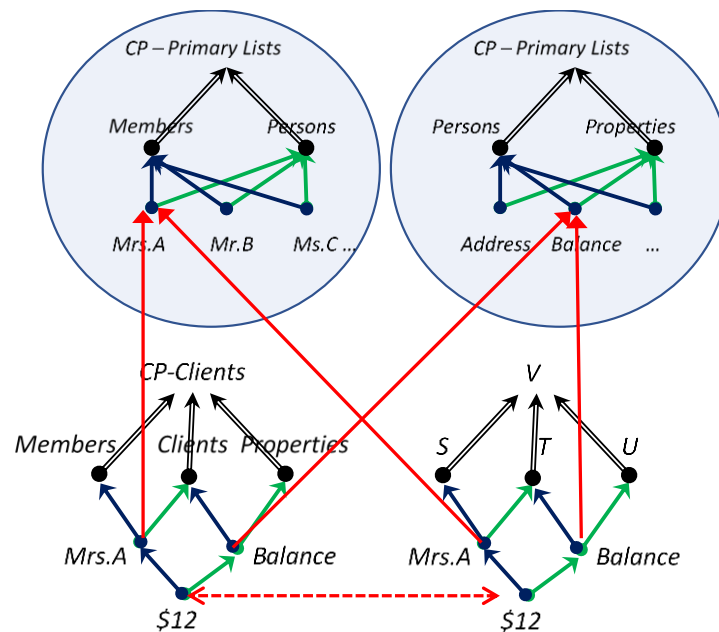


Figure 3.2.19. The two “\$12” facts at the bottom are distinct facts, with distinct facts and propositions in their interiors, but they have *identical first-order propositions*, because the three terms in the triads (*Mrs. A*, *Balance*, *\$12*) have the same referents in both cases.

Note the referents for “*Mrs. A*” and “*Balance*” tokens are determined by the 3rd joins. The referential identity of the two “\$12” values is determined from the numeric representation directly. This is the special case of *numeric identity*. Numbers are given *unique names* (through number format fields), and we do not need to represent numeric identities through 3rd joins.

Note also the first order propositions are independent of the points *R*, *S*, *T*, or *Members*, *Clients*, *Properties*. The *first order propositions are weaker than the facts*, as there may be several distinct fact tokens that state the same first order proposition.

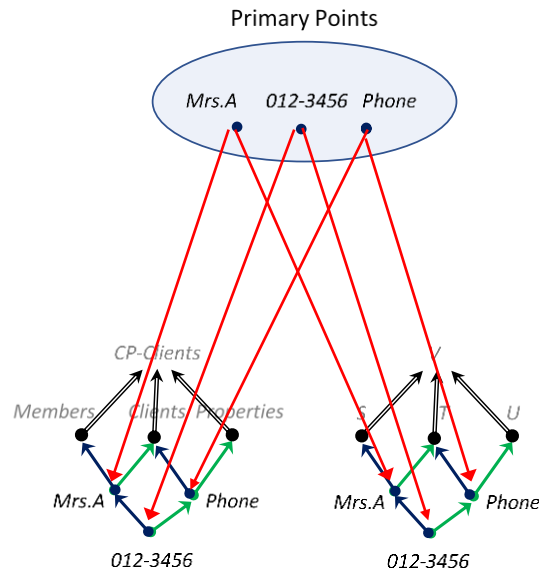


Figure 3.2.20. Two identical first-order facts. Seen from the point of view of the *primary points*, we can identify three reference points we want, and look down their reference trees for intersections, where the points are in the triadic relation.

Two such triadic relations, with the referents in the same order, represent *the same first order propositions*. This proposition may be repeated in a network in several places. To find these instances, we just take: $(Mrs. A) \vee (012-3456) \vee (Phone)$. To find all the values associated with *Mrs. A* as *phone numbers* we can take: $(Mrs. A) \vee (Phone)$. (We will see later it is very efficient make such queries on the CAT4 table.)

Appendix 3.2.1. CAT3 Propositions and facts.

We explain the interpretation of *propositions* more technically. Each *point (fact token)* has a *first order proposition* associated with it. This is the proposition constructed from the *referents of the point and its two parents*.

- Definition. The first-order proposition of a point *p* with parents *q* and *r* is the proposition constructed from the *referents of p,q,r*.

E.g. “Mrs. A has a Balance of \$12”. Logicians could symbolise this as a logical function: $B(A) = C$. This takes *Balance = B* as a *function mapping persons to values*. This function is instantiated in CAT2, with points for *Mrs. A = A* and *Balance = B* adjacent, and their child points for values. The *function B* applied to *A* results from taking: $A \vee B = C$, so the function $B(.)$ may be defined by abstraction. Using the Church lambda calculus:

- $B(.) = \lambda A.A \vee B$ Predicate function of B

Note that this construction: $B(.)$, which is a *function of B*, is not the entity *B* itself. It is a *construction from B and the points adjacent to B in the graph*. These points represent contingent information: hence the function: $B(.)$ represents contingent information. The graph gives a space to represent such functions. We can construct a more specific *first-order function*, by defining the operator: $A|B = C$ just in case *A* is the left parent of *C* and *B* is the right parent. This returns only the immediate children of *A,B* (which is the intersection of points: $A.[1,0]$ and $B.[0,1]$).

- $B(.) = \lambda A.A|B$ Left predicate function of B

This should be distinguished from the general function $B(.)$, and we may write typed predicates:

- $B_{[0,1]}(.) = \lambda A.A|B$ Left predicate of B, typed
- $B_{[1,0]}(.) = \lambda A.B|A$ Right predicate of B, typed

This gives an array of *typed functions*: $B[x,y](.)$, with *x, y* any finite integers. Outside the limits of the exterior and interior of *B*, such functions become null. We can show that a set of such types covers the entire range of functions defined by CAT2. Note also in this language, we can abstract on *x* and *y*, and write a typed predicate function for $B(.)[.,.] = \lambda x \lambda y. B[x,y](.)$, and abstract again on *B* for a general predicate function, etc. The most general function however is just the exterior meet: $\vee(.,.) = \lambda B \lambda A. A \vee B$.

Note that equally, we may abstract the symmetric function:

- $A(.) = \lambda B.A \vee B$ Predicate function of A

which now treats a *person* (Mrs. A) as a function from properties (Balance) to values (\$12). The system is formally symmetric with respect to *A* and *B*. We may write: $A(B) = C$ or: $B(A) = C$. And: $A(B) = B(A)$. We must remember that $A(.)$ is not *A*, and $B(.)$ is not *B*.

We now define the *first order proposition of a point, p*, as distinct from its identity as a fact, which is dependent on the entire interior network. As a first attempt:

- The *first order proposition* of a point *p* with parents *q* and *r* is the proposition that: $q(r) = p$, or: $r(q) = p$, where p, q, r are interpreted appropriately as predicates and subjects and values.

The essential point is that this proposition involves *only the immediate referents of the points, p, q, r*. The parents of *q* and *r* or other aspects of the construction of *p* do not matter. This immediate proposition simply combines the *real-world referents of p, q, r into a proposition*, just as with an ordinary proposition (*however you think this is done*). We may symbolise this initially as:

- $P = Prop(p, q, r)$

Note in this equation, the function **Prop** acts to combine the *referents of the terms, p, q, r*, into the proposition about them. In CAT3 we can identify the common referents of all propositions as the primary points, defined by the 3rd join. We may say that: *the first order proposition about p* is equally a *proposition about p', p'', etc, i.e.* all points in the 3rd join tree up to the *primary referent of p*.

- We may use the notation: 0p to identify the *primary referent of p*.
- The first order proposition made by the fact token *p* is:
- $P = PROP({}^0p, {}^0L(p), {}^0R(p))$

where $L(p)$ and $R(p)$ are the left and right parents of *p*.

In Fregean terms, this distinguishes *sense*, which is the relationship of the term *p* to *q* and *r*, and *reference*, which is 0p , as the two essential components of meaning of a term *p* in an expression: $p \varepsilon q \vee r$, or: $p = q \vee r$. A term *p* (fact token) has a *sense* only in a sentence; it has a *reference* as a constant property of its *symbolic interpretation*. This is constant if we insert the *fact token, p*, in different places in the graph.⁶

⁶ Note our 0p operator here corresponds to an analogous construction, *trivialisation*, in Tichy's TIL [1987].

3.3 CAT4 Semantics: time, truth, functions.

We now extend CAT3 to include *time*, *truth*, *functions*, and *content*, giving the CAT4 relation.

CAT4 Fact Table.

There are two universal features of propositions we have not mentioned yet: they hold over *times*, and they have *truth values*. We can state propositions as being *true or false*. We can state them as having a value in a certain period, e.g. *Mrs. A's Age is 50 from 9/10/2018 to 8/10/2018*. On the 9/10/2019, it changes to 51. These properties of *time* and *truth* could be added as *exterior values*, but because they are universal that would be mean adding these as external properties throughout the graph, and it would be impossible. Instead we characterise them as *properties intrinsic to facts*, making facts intrinsically *temporal*, and intrinsically *truth-valued*. We also extend the earlier CAT3 to CAT4, with a new join, *ID4*, which will represent *functions*.

CAT4 Fact Table.

Logical	CAT4 Relation				Time		Truth	Content	
ID	ID1	ID2	ID3	ID4	Time1	Time2	Truth	Name	Value
0	0	0	0	0			1	Zero Point	
1	0	0	0	0			1	Members	
2	0	0	0	0			1	Clients	
3	0	0	0	0			1	Properties	
4	1	2	0	0	08-08-18		1	Mrs. A	
5	1	2	0	0	01-07-17		1	Mr. B	
6	1	2	0	0	23-09-16		1	Ms. C	
7	2	3	0	0			1	Full Name	
8	2	3	0	0			1	Age	
9	2	3	0	0			1	Balance	
10	4	7	0	0	08-08-18		1	Ann A	
11	4	8	0	0	09-10-18	08-10-19	1	50	50
12	4	9	0	0	09-10-18		1	12	12
13	5	7	0	0	01-07-17		1	Bob B	
14	5	8	0	0	03-07-18	02-07-19	1	35	35
15	5	9	0	0	09-10-18		1	5	5
16	6	7	0	0	23-09-16		1	Carol C	
17	6	8	0	0	16-04-18	15-04-19	1	28	28
18	6	9	0	0	09-10-18		1	7	7

Figure 3.3.1. A table representation of the primary CAT4 fields, used to characterise points as empirical facts, supporting a temporal logic of propositions and functional constructions.

- *Exercise: When is Mrs. A's birthday? What was her Balance on the 8/10/18?*

We have introduced the 3rd join, the 4th join is quite similar, and we discuss it later. But we are concerned here to explain the extra columns. These should be intuitive to start with.

- *Time1* and *Time2* represent the duration of the fact, i.e. a start and end time.
 - Nulls may be taken to indicate the indefinite past (in *Time1*) or future (in *Time2*).
 - For a full temporal logic, we also need a symbol for the present moment (*Now*).
- *Truth* represents the truth value of the first order proposition of the fact.
- The *Content* fields carry symbolic data, names or values. These normally have referential meaning in natural language, or in the meta-language of the data system.

We will return to the *Content* fields later, but their logical purpose is clear enough. They record *symbols associated with fact tokens*, and these generally provide reference (in an external language) to real world entities, such as individuals, properties, values, numbers, files, functions, etc. Functions generally act on *content of tokens*. E.g. numerical functions like: =*Sum* or: =*Average* act on numeric values found in the *content* of tokens. We have displayed such values in the *Names* of the fact tokens illustrated above, and this is the primary, visible *content field*. However there are several *logical content fields* for fact tokens, including (i) a *name* (text field), (ii) a *value* (real value or double-precision number), (iii) a *date-time interval* (two date-time values: *date1* and *date2*), (iv) an *order* (integer value), (v) a *truth value* (0/1 or real value, for epistemic probabilities). We think these are adequate for a complete propositional system. Numerical functions generally work on the *value* field. The *value* may be copied to the *title*, but not necessarily.

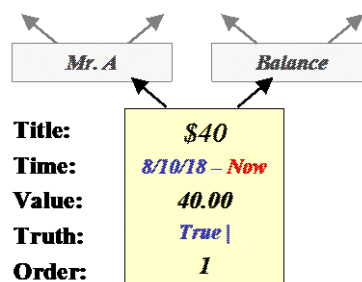


Figure 3.3.2. A CAT4 fact token with all its *content fields*.

CAT4 Time and dated facts.

To represent facts as intrinsically temporal, as part of the *logical content* we associate each point with a *time interval*. For this, we add two *date-time fields*: *Time1* and *Time2*, as part of the *logical content fields* of points. This is distinct from reference to ordinary entities (individuals, etc). Time

reference is to a universal linear time. Time is used in an intuitive way. Facts are stated to hold for certain intervals of time, from *time1* to *time2*.

A key feature of CAT4 is that we record *time series* by using multiple *dated fact tokens* in the same category, e.g. *Mrs.A/Age* or *Mrs.A/Balance* will typically contain a *stack of dated fact tokens*, representing the changing facts. For properties like these with unique (but changing) values at each time, the stacks should record a time series that does not overlap. But generally time series can overlap, and properties or categories are multi-valued at moments of time.

Two fact tokens with identical parents and content therefore *represent the same fact*, but they may state it as holding *at different times*. Our *intervals of time* can include single instants, by letting *time1* = *time2*, and the indefinite past or future (e.g. using nulls as a convention in a table). All temporal facts and temporal relations can be systematically represented in this way, as long as we have symbols for *the indefinite past*, *indefinite future*, and the moment *now*.

- Note this system of having a single universal time for all facts replaces layers of data programming in RDBs, where time is treated as an extensional property.

Note we normally want *the primary point* for an entity to be valid for its entire history. The interval may be taken as the period of 'existence' of an entity, but this must be the period in which it is a subject of facts, not its physical life-time. This may generally be taken as timeless. There are facts about things before they exist and after they cease.

By representing every individual fact as intrinsically temporal, we can represent all the temporal relations among facts. The atomic facts are *atoms for temporal propositions*. We can also represent temporal facts *extrinsically* sometimes, as properties, as we saw for *Truth Values* for instance. However there is a common temporal fact we refer to in real life that we have not yet characterised in CAT4, namely, *The Present Time*. We call this the *Now*. This refers to an essential metaphysical feature that must be incorporated into any real world semantics, which is *time flow*. We need to associate facts with *times*, to reflect the changing nature of empirical propositions. The empirical world is temporal, and facts change with time. Some facts may be eternal, but we can take this to mean they *hold at every time*, not that they are a-temporal.

Note the CAT4 representation itself (the database table) is temporal, and changes with the *present moment*. It goes through a series of states, and generally expands with time. It seeks to represent *the facts as they are known now* at the moment *now*. The CAT4 dated records represent a record back in time, of the database state at times. So it is like a sequential movie of the database states,

through time, up to the present state, and this is contained part of the present state. This is *memory*.

- This structured temporal state, or *memory*, in the representation correlates to *time flow* in the real world.

So it is necessary to characterise facts about the *present moment*, or *Now*, in our system. There is a computer function to do this, the system clock function: `=Now()`. This is an indispensable function in computer systems.

- `Now()` has no arguments, and generates a date-time symbol, the name of *a specific time*, whenever it is activated (“called”).
- The value it produces is *the name (i.e. date-time) of the present time* when the function is activated.

We must introduce a special term to represent *Now* in the time variables, *Time1* and *Time2*. We need to introduce a symbol so we can write *current-past-intervals* like: `[2021, Now]`. This means: *the interval from 2021 to Now*. And similarly, *current-future-intervals* like: `[Now, 2025]`. This means: *the interval from Now to 2025*. E.g. we would write: *Joe Biden is president of the USA [2021, now]*. Or: *Joe Biden’s first term as president lasts from [now, 2025]*.

- In this symbolism, “*Now*” effectively means the clock-function: `=Now()`.
- The time variables are real numbers. For a special symbol, we could define: `0` or `-999` to mean *Now*, but actual times recorded as: `0` or `-999` would no longer be valid. We need to find time values `0` or `-999` and treat them specially when processing time data.
- We suggest using `null` for the *open future* in *Time2*, and the *open past* in *Time1*.

The function: `=Now()` obviously *changes its value* as time goes on. But the function does not change its *meaning* or its *programmed process*. It always means: *the present time*. What the *present time* is is a *contingent matter*, shown by the fact that the computer clock may be wrong. E.g. it may be printing out *the time five minutes before the present time*, if it is running slow. We can check its accuracy against other clocks, as there is an objective standard for our time references.

It has been and still is popular among philosophers discussing time (from Russell to Reichenbach, and beyond) to hold that the *meaning* of the term “*now*” is changing, and that tokens of the term “*Now*” produced at different times thus have different *meanings*. This so-called “*token-reflexive*” theory of meaning for *Now* is very popular among philosophers, but it is wrong. The *meaning* of the term “*Now*” does not change. The *value* of *Now* changes. In a computer program, it is referred to the

fixed function: $=Now()$. In the intensional logic of TIL, it is similarly referred to a fixed function: $Now(World, Time) = Time$. This function does not change. It is the *value of the function, the value of the present moment*, that changes. What is presented to us in real life, to evaluate the truth of propositions against, is a *world-at-a-time*. We call this the *actual world, now*. It is not a timeless, 'bloc universe' world, with no *present*.

We cannot deal with *changing values of terms or propositions* by saying the *meaning of the term or propositions changes*. Otherwise we would also say, e.g. that *the meaning of "the temperature"* changes every time the temperature changes. We would have to know what the temperature is to know the *meaning* of the term. Similarly, on the indexical theory, we would have to know *what the present time is* to know the *meaning of the term "the present time"*. This is an incoherent theory of meaning, and has been criticised repeatedly in the past.

CAT4 Truth Value.

The *truth value* is more ambiguous to interpret. We identify it as characterising the *first order proposition*. There are other propositions associated with facts (the propositions of their interior facts), and these also have truth values. But these do not affect the truth value of the first order propositions. So our basic principle is:

- The *first order propositions*, constructed from the referents of the three points, are atomic, and they are what we characterise intrinsically with the *fact as true or false*.

Note that this means the truth value characterise the *propositions*, rather than simply the facts. We might take it that the special function of a fact network is to support propositions, and this is an intrinsic feature of the fact interpretation. Truth value associated with *facts* supports the required logical relation between *facts* and *propositions*. Higher-order propositions are constructed from first order propositions, reflecting ordinary logic. We adopt at least a three-valued truth set.

- Every point is assigned a Truth Value (TV), from: $\{1, 0, -1\}$, representing *True, Unknown, False*.

This method will allow us to treat negation by reversing the truth value of the whole proposition, the treatment recognised by Frege. Propositional logic requires *propositional negation*, which is provided in CAT4 by assigning the *Truth Value* of fact tokens as *False*. (Or more generally, reversing the truth value). Thus our *fact network* is overlaid with *truth values*, to represent the presence or absence of facts explicitly, and to state propositions as true or false explicitly. It essentially provides a Boolean algebra over the primary CAT2 lattice, and the representational power of CAT2 when we add this *binary truth predicate* is greatly enhanced.

We first explain why we need this interpretation.

False Facts and Propositions.

The basic conundrum with interpreting *truth value of facts* is seen from this example.⁷

etc	Etc	etc	etc
Members TRUE	Clients TRUE	Properties TRUE	
	Mrs. A FALSE	Balance TRUE	
		\$12 TRUE	

Figure 3.3.3. This CAT2 tablet shows points with *Truth Values*. Here a True proposition (the “\$12” token) is stated under a False proposition (the “Mrs. A” token). Is this consistent? Should we take points below *False* facts to also be *False*?

The *Mrs. A* fact token is labelled *false*. This means that *Mrs. A is not a Member of Clients*. But the \$12 point below is labelled *True*, and states a *true first order proposition*. (We assume Mrs. A does have a Balance of \$12.) Is it sensible and consistent to represent *true facts* under *false facts* like this? Surely *false facts* are false because they do not exist in the world, so how can true facts be defined in relation to them? However, the fact that *Mrs. A is not a Client* does not mean *Mrs. A does not exist*. We assume that the *False* fact token for “Mrs. A” still refers to Mrs. A, as before. To clarify, we can state the same proposition, using only *true* facts, e.g. as follows.

etc	etc	etc	etc
Ex-Members TRUE	Clients TRUE	Properties TRUE	
	Mrs. A TRUE	Balance TRUE	
		\$12 TRUE	

Figure 3.3.4. We have now characterised Mrs. A as an *Ex-Member of Clients*. She was a Member, but ceased to be one. She still has a Balance of \$12.

⁷ When we were illustrating multiple join structures, we used the point-graphs, but when illustrating single propositions or facts we can use flat CAT2 tablet diagrams. Remember there may be multiple fact tokens in any position. We are generally only interested in the local graph around a point of interest, and can ignore points outside our focus (“etc, etc”).

This makes sense, and the two first order propositions represented by the two “\$12” tokens in the two graphs are identical. Thus, in the previous diagram, we have a *true proposition stated under the assumption of a false fact*. And we may have further facts below that.

- False propositions in the interior do not make propositions in the exterior false.
- Truth value is not inherited through the interior-exterior fact relations.

Note there are many practical situations where we want to record *facts as false*. E.g. we might get into this situation by first recording *Mrs. A* as a *Member of Clients*, as a true fact, and accumulating information about her in this role, but later altering this to a *false fact*, when *Mrs. A* becomes an ex-client. Thus we may be left with true facts recorded under false facts.

Now we may object that this treatment of the changing fact about *Mrs. A* is not really an accurate representation. We should instead *end the current record of Mrs. A. as a Member of Clients at a certain time, recorded as Time2*, and start a new record for her as an *Ex-Member*, with the start time as *Time1*. This way, all the records remain *True*, and the historical record remains true. (It can be problematic to change the truth value of a record *retrospectively in time*, but we can define general methods in CAT4.) This strategy avoids using False facts, and may make us wonder:

- Is it generally possible or practical to manage a representation so that we do not use *False facts*, and instead use only *True facts*?

The short answer is: *for simple data, Yes, but for general information, No*. We can usually translate ordinary database records into CAT4 records as positive facts, but this is because the RDB method itself is generally limited to positive facts. It provides no intrinsic way to represent *propositional negation*. We can construct representations of negative predicates, or add ‘truth value’ columns to tables to characterise rows as “True” or “False”, but this is treating negation as an exterior predicate. This does not work satisfactorily in RDBs any more than in CAT4, and leads to a proliferation of *ad hoc* predicates and columns, that must all be programmed. It is unscalable.

Note the basic SQL logic (c.f. Codd’s theorem) assumes tables provide a *complete record of the atomic facts of their relation*, so that absence of a record of a fact in a table implies the *negation*. E.g. *Mrs. A does not have a red car* is inferred if there is no record of a red car owned by *Mrs. A*. But with partial information tables, we can make no inference. Instead, we need to be able to represent the fact that *Mrs. A does not have a red car* directly, not by making a complex secondary inference.

- For a general information system, we must be able to represent fact negation or propositional negation generally, by characterising facts intrinsically with *truth values*.

Moreover, in our present example, although it may be possible to re-organise the information to avoid using *False* facts, it may be more natural and desirable to characterise some facts as *False*. E.g. suppose we may have recorded information about Mrs. A over a period of time, under the mistaken belief that she was a Member of Clients, and subsequently discovered that she never was a Member of Clients – lets say, due to a legal technicality that no one noticed earlier. Records for Mrs. A also include various Client-Property records, such as Balance, Age, etc. Now to correct our error and re-establish *true past records*, we can retrospectively change the fact of Mrs. A's Membership to *False*, while retaining her other records without change, throughout the period when she was treated as a client. This makes sense semantically, it makes the records true and consistent, and CAT4 gives the power to represent this.⁸

We now illustrate this further with our CAT3 relationship graphs.

Propositions in relationship graphs.

Returning to our earlier example, the same *proposition* appears to be represented at the bottom on either side of the graph below: “The son of A is B”, “Of A, B is the son”. We will take these as analytically identical. How then do we represent this identity? Can we identify them via the 3rd join?

⁸ For another alternative in this situation, we could *move* the left parent of Mrs. A to a *Non-Members* point instead. But this requires we have a suitable ‘negative category’ defined (“Non-Member”), which must be logically programmed (so we do not have the same person as a *member* and *non-member*). We cannot individually program negative versions of *all predicates (columns) and their property values* in an RDB. There are thousands, and they are open-ended. E.g. to characterise Mrs. A's car as *not red*, we could add a *not-red color predicate* (e.g. in a look-up table for *color values*), but to make negative versions of all properties systematically, we must duplicate multitudes of property values and the required logical functions for each one. This is impossible. Also, changing a parent changes the meaning of facts dependant on the fact of Mrs. A being a *Member of Clients*, e.g. the *count of members of clients*, the *total balance of members of clients*, etc. It is tricky to *move fact tokens* around in the network.

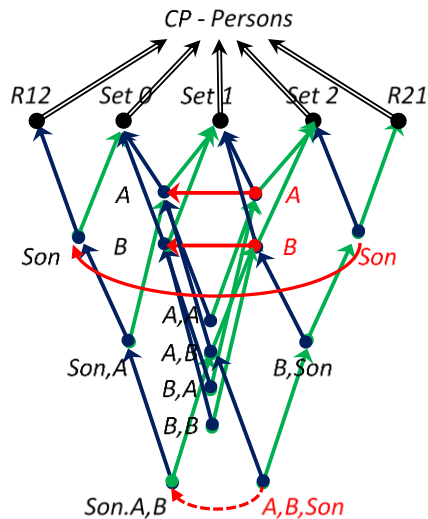


Figure 3.3.5. We have identified the two points labelled “Son” by using the 3rd join, meaning they have the same *referent*. Similarly with “A” and “B”. We have taken the left points as primary, and the right as secondary. Can we similarly identify the two points at the bottom? What *entities* do they represent?

The empirical referent of “son” is the real-world: “... is a Son of ...” relationship. This is applied to ordered couples: $Son(A,B)$ means: *A is a Son of B*, while: $Son(B,A)$ means: *B is a Son of A*. We can apply *Son* to an ordered couple, (A,B) , from left or right in CAT3 – both give the same class: $Son^{\vee}(A,B) = (A^{\vee}B)^{\vee}Son$. We distinguish these in the diagram by writing: Son,A,B and A,B,Son for the points at the bottom.⁹

These points must represent identical propositions, but what do they mean? What are their *referents*? One answer seems to be that they *mean the propositions*, respectively: “A Son of A is B” and: “Of A is B a Son”, which are logically the same proposition with a trivially different construction. So we should identify them?

But what *entities* are we identifying? *Propositions* have *truth values*. They are treated by intensional logicians as mappings from *worlds*, or *world-times*, to *truth values*. Their distinguishing feature is that they may be *true or false, of worlds*. And they may be stated to hold of counter-factual worlds, of which they are true, while being false of the actual world.

⁹ We assume a consistent system for representing relationships, so e.g. “... is a Mother of ...” will have the same formal properties as “... is a Son of ...”. Hence: “The Mother of B is A” will be a point below the pair (B,A) , either as: $(B,A)Mother$ or: $Mother(B,A)$. See next section for the more detail of the symmetry.

Hence we might characterise our *propositional facts* with truth values, assigning them the value: *True*. Compare with recording ordinary property-value facts, e.g. “the Distance from A to B is 3.14 meters”.

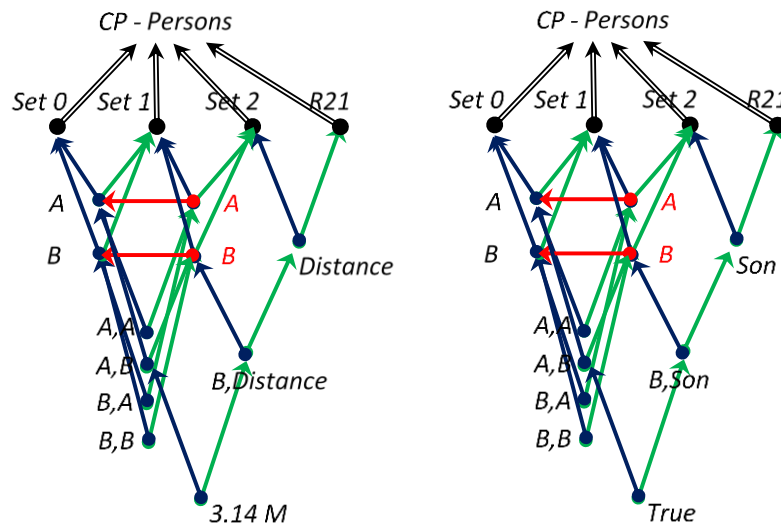


Figure 3.3.6. Left, this is a natural way in CAT3 to record a quantitative relationship, like distance from one person or place to another, A to B. The value: “3.14 M” has parents “A,B” and “B,Distance”, and GP’s “A”, “B”, “Distance”. Right, since propositions have truth values, if the points representing: “a son of A is B” or “of A is B a Son” are propositions, they may be analogously characterised as *True*.

In this way we see how *truth values* can be taken as *extensional values*, *characterising relations*, analogous to *distances* as *extensional values*, *characterising relations*. However we already characterise facts intrinsically with truth values, so *propositions* may need to be treated differently. Before discussing this further, we clarify the concept of symmetric copies of information.

Symmetric copies.

The symmetric copy on the left above is a feature of the symmetry of the representation, viz that if we can logically apply a concept on the right, we can define it to apply symmetrically on the left.

However applying the left *Son* relation gives the intermediate function: *Son of A is ...* , applying the right *Son* relation gives: *Son of ... is B*. These are distinct.

This is seen in terms of an abstraction operator, operating on the left:

$$(B).Son \equiv \lambda B \lambda A (Son(A,B))(B) \rightarrow \lambda A (Son(A,B)),$$

and operating on the right:

$$Son.(A) \equiv \lambda A \lambda B (Son(A,B))(A) \rightarrow \lambda B (Son(A,B)).$$

We see points operate (the *Join product*: $A \vee B$) left or right symmetrically. It appears there are two functions: $\lambda A \lambda B (Son(A,B))(A)$ and: $\lambda B \lambda A (Son(A,B))(B)$, associated with the two *Son* points, but we may take it as a convention instead that there is just one function, and the change from a left to a right *application* has the effect of *reversing the order of functional application*.

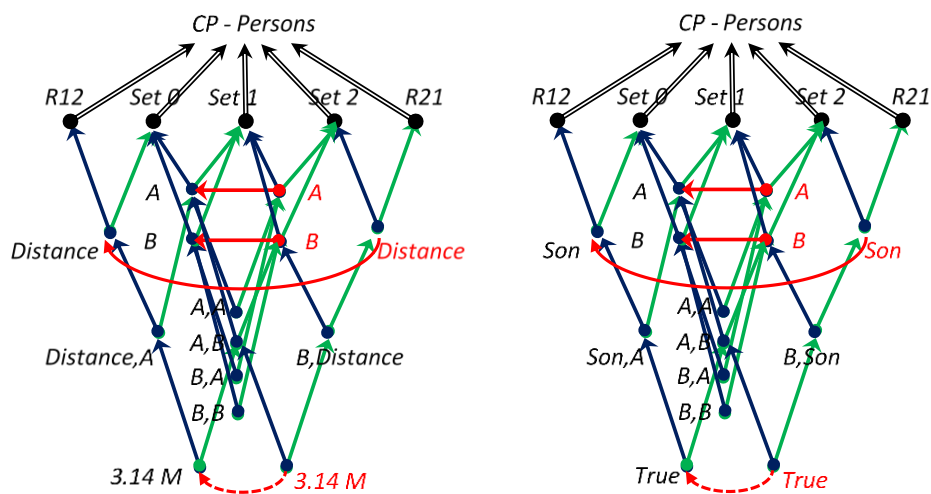


Figure 3.3.7. This illustrates a parallel between property-value facts and relationship-truth-value facts. Note the bottom pairs on either side have identical referents and propositions.

This appears as a correct way to represent *extensions of the properties*.

Truth-functions and propositions.

However we now seem to have a potential regress of truth values.

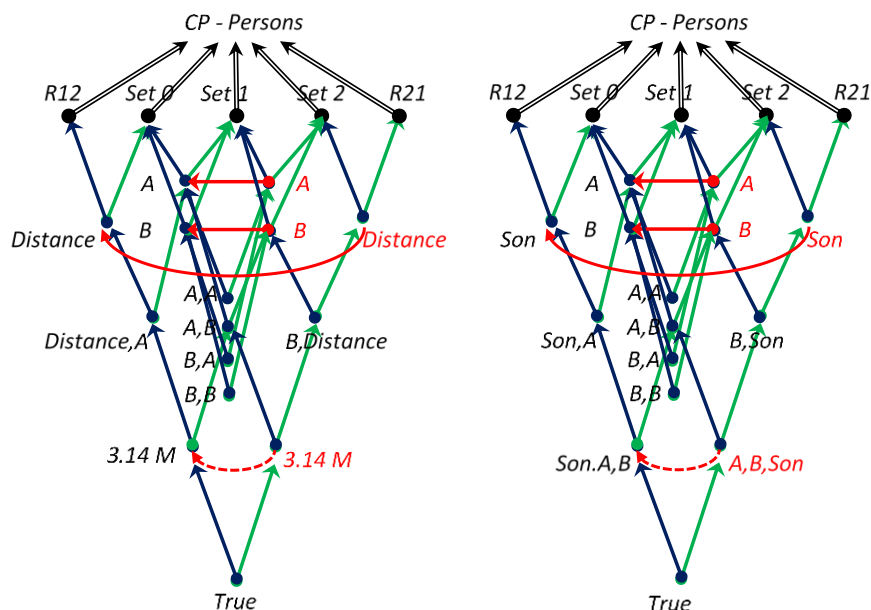


Figure 3.3.8. Here we use dual (symmetric) representations. Left, since “3.14 M” represents a proposition, why not also assign it a Truth Value, *True*, as an extensional property? Right, if the complexes of: (Son,A,B) or: (A,B,Son) denote *propositions*, should we represent *propositional symbols* for these, and assign these Truth Values below, as properties?

But if we assign these exterior truth values, can we assign further truth values to them? And in fact does it make sense? The potential regress of *truth values* is stopped because *every point in the CAT3 fact network is assigned a truth value intrinsically*. However the example above shows the potential confusion when a *truth value* is really the (*extensional*) value of a data point.

- On the left, 3.14 is the value of the data point, it has an intrinsic TV as a fact, and the second point below saying it is *True* is redundant.
 - If the TV of the 3.14 point is *False* instead of *True*, we must still take the exterior truth value to be *True*. The whole proposition is: $(3.14 \text{ is the value is false}) \text{ is true}$.
 - We can never assign *False* to the exterior value without producing a contradiction between it point and its parent. For they state contradictory facts: one states a proposition, the second states that the first proposition is false.

The exterior method of assigning truth to facts cannot work, because it requires we state a fact first, and then assign it a truth value.

- On the right, the *value* of the data point we have labelled “*Son,A,B*” is *True*. We have represented *Son* as a binary function over pairs (A,B) , and this has *True* or *False* as its values.
- The second *True* point below is also redundant, as in the first case.

However this suggests that we could possibly interpret the points “*Son,A,B*” and “*A,B,Son*” as naming the *propositions*, not just the truth values.

- The function that produces the values is equivalent to a propositional function, evident simply because it generates TV’s as the extension. *Son*: $(A,B) \rightarrow \text{Truth Value}$.

So what if we take these points: “*Son,A,B*”, and “*A,B,Son*”, as *constructing propositions*, and give them exterior values, *True* or *False*, with child points below, dated at different times?

Then we are not just representing propositional extensions (true/false), but listing *propositions* themselves. However what is the *first order proposition* of this “propositional fact”: “*Son,A,B*”? It is saying that this *proposition* is the result of combining the two parent points with the fact point. The parent points are *Son(.,.)* and (A,B) . If we combine them functionally as usual, we get the truth value. But now we are proposing to get something else: the *first order proposition*. This is the first order proposition *formed from combining the Truth Value of the point with the parent entities*. I.e. the *fact TV* is being taken to provide the *fact referent*.

To make this clearer, if *P* has parents *Q* and *R*, then the proposition is that: *Q of R is P*. But *P* in this case has the value of a *truth value*. So we imagine we use its intrinsic TV for this, and assign it a *second value*, viz, *the proposition that Son(A,B) is true*. Thus we may take it as an abstraction over its own truth value, and over the *construction* of the proposition.

But this would mean we have changed the formal interpretation of the network joins – using it first to represent *functional application*, but now to represent some form of *propositional abstraction* for the special example of propositions. In fact this is a second-order *construction*. But this is NOT how we will treat the propositions. It is not consistent to sometimes use the fact interpretation one way, then another. If we want to “abstract” propositions from their facts, or extensions, we must apply the abstraction through a function.

This shows how confusing it can be to try to think through the semantics for propositions, and the choice of model for natural language propositions is the critical point for applying CAT4 to language. We will develop a system for transforming *NL Sentences* into *CAT4 Propositions*. And we will need to represent both sentences and propositions in relation to each other for this. We adopt a quite

different method to the previous idea for propositions. And this will give us a CAT4 model for natural language.

So far we have found a number of representations for different forms of structured data and relation. We see there are alternative ways of using these, but not many, and they are all highly structured and inter-translatable. E.g. we can map: *Tables* → CAT4, *Graphs* → CAT4, *Hierarchies*, etc. But the most difficult communication form to translate is the major one, i.e. natural language, *NL*, and no one has found a general method for *formally translating sentences into information* yet.

However before we propose a natural language representation, we have one more fundamental concept to introduce. We need *functions*, which are provided by the 4th join.

CAT4 functions.

Here we briefly introduce *CAT4 Functions*, and discuss them at greater length in Part 4. Functions are required so we can calculate logical-mathematical consequences of existing facts, and represent them in the CAT4 graph as further facts. We do this with calculated fields in database tables, and calculated cells in spreadsheets. The spreadsheet is the best example to visualise, because *every cell can be assigned its own function*. We enter *function strings* in cells, e.g. “=[A1]” in the cell [B1]. This will replace the value of the cell [B1] with the value of [A1] when it is executed. Or: “=Sum([A2:A6])” in [B1] will enter the sum of the cell range in [B1].

Simple functions return single values, and enter them as the cell value in the cell containing the function. More complex functions can take a range of values in a *source location*, calculate a new set of values, and insert them in a *target* location. More complex functions still are general programs, and perform simpler functions in sequence, manipulating cells properties, updating values, etc.

Note spreadsheet cells have two separate properties: *value* and *function*.

In CAT4 we similarly assign *functions* to points. Atomic data points usually do not have functions, but *calculated fields*, or any type of *logically dependant points* are represented by functions in the CAT4 graph. We may imagine first that for every CAT4 point, we can enter one function, in a format like:

$$= \text{Function}([\text{Source_Range}],[\text{Target_Range}]).$$

We can reproduce all the common functions found in spreadsheets in CAT4, and the CAT4 coordinate reference system is similar in form to spreadsheet coordinates (just having the extra “^” and “v” operators), so we can adopt this syntax in CAT4 quite literally. We will define a functional

language like this. However although this is a valuable tool for us, it is not our primary representation of functions.

How should we represent functions associated with points? We have two content fields, for *Names* and *Values*, but we cannot use these for *function strings* because they already have other tasks. Anyway we do not simply want to record functions as function strings: this is the conventional representation, but it leaves the construction of the functions opaque as *information*. We need a transparent representation of the *facts about functions*. This must be represented in the CAT4 graph. This must cohere with our semantic interpretation generally.

We introduce the 4th join to represent *the function of a point*. It has the dynamic aspect, that functions may be *executed*, they are procedures that alter the graph, and they are distinguished from the three other semantic joins. The 4th join directly associate a point with its *function token*. The facts about the function are stored under this point. This is the general semantic principle.

- If S is the 4th join of a point P , then S represents the *instantiation of the function of P* . It is called the *function token for P* .
- The right parent of *the function token* represents the *function*. These are the *primary function points*.
- The *exterior* of the function token represents its *parameters*.
- The *function engine* calls functions through the *function points*.
- The conventions for interpreting *functions and parameters* are defined in the function engine code.
- The *function token and its exterior* must be sufficient information to determine the functional execution through the function engine.
- One function token may be joined to by many function points.
- When a point function is executed, it executes all the functions in its 4th join interior, up to its *Primary Function*.
- When a function token is executed, it executes all the functions in its CAT2 exterior.

Summary function examples.

We need to be able enter fact tokens to represent facts about *totals*, *averages*, and other ‘calculated fields’, such as we generate in spreadsheets and queries. There are many more functions, but we start with this basic type.

Sum	\$90	\$107	\$114
Avg	\$30	\$36	\$38
Client	Balance	Previous Balance	Max Balance
Mr. A	\$12	\$5	\$12
Mrs. B	\$24	\$48	\$48
Ms. C	\$54	\$54	\$54

Figure 3.3.10. A spreadsheet grid with some primary data (blue) and calculated data (red). In the spreadsheet, the calculated cells (red) have *functions* entered, e.g. “=Sum(B4:B7)” gives the sum of “\$90”. We see the result of the calculation in the cell. (The simplest function is just another cell reference.)

To read the grid of data above, we must use our experience to infer that the top row refers to the *sums of the blue columns below*. We use our experience to infer that the right column refers to the *maximum value of rows (Balance and Previous Balance) for each client*. These readings reflect *informal semantic conventions* for reading tables of data. In CAT4, these semantic relations are made explicit. CAT4 uses *4th joins*, to *function tokens* to specify functions, and *function parameter tokens* below these to specify parameters. These *parameter tokens* use *reference joins* to refer to the *source fact tokens* that we need to calculate from – and to the *target fact tokens* where the function writes its results (not necessarily the token defining the function). *Functions* are also categorised in a more structured way than in a spreadsheet.

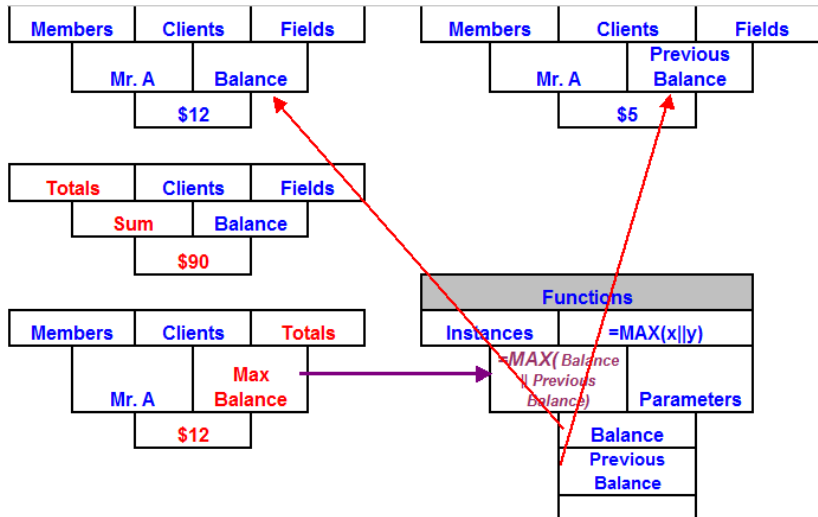


Figure 3.3.11. Example of a function. Data at the top we may consider ordinary 'table data'. Below that the *Sum* function is shown inserted on the left. Below that the *Max* function is shown inserted on the right. The functional 'wiring' for the *Max* function is shown. The wiring for the *Sum* function is similar.

The *Sum* and *Max Balance* header tokens both have *function joins* to instances of functions - this is how the system knows how to calculate them. The *target tokens*, where the calculations are written to, do not necessarily have function joins. However we can *join target tokens by function joins to the function definition token* that produces their values if we wish, as illustrated below. The function execution will create such joins. This can disambiguate results if more than one function points to the same target.

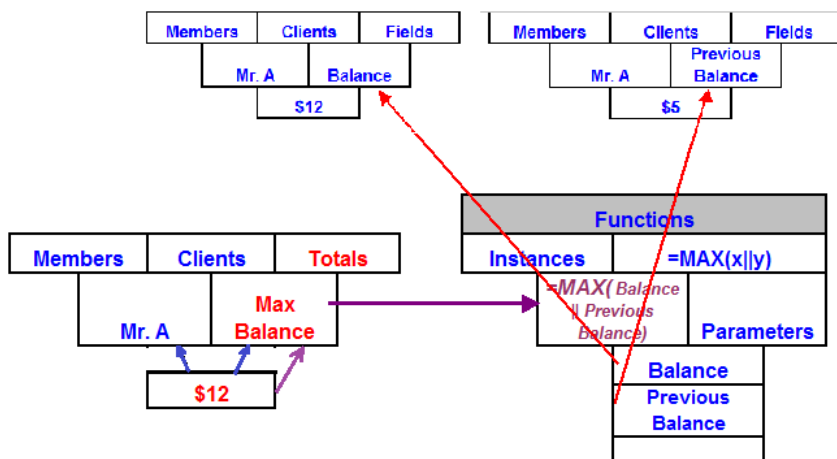


Figure 3.3.12. The function value target (\$12) is here shown joined to the function definition token (*Max Balance*) that generated it, by its function join (purple). This means: \$12 is calculated by the *Max Balance* function.

Category parents for calculations.

It is only necessary to get the *category parents* right to insert calculated fields. To insert the ‘Sum’ function (which adds *down the Client-Members list*), we have inserted a new category parent ‘Totals’ on the left – so we do not get these ‘Totals-for-Clients’ records mixed up with the ‘Members-of-Clients’ records. We may insert any 1-place function in here, e.g. *Sum, Average, Count, etc.* To make this more explicit, it corresponds to ‘grouping’ the tables rows like this:

	Clients	Balance	Previous Balance
Members:	Mr. A	\$12	\$5
Members:	Mrs. B	\$24	\$48
Members:	Ms. C	\$54	\$54
TOTALS:	Sum	\$90	\$107
TOTALS:	Avg	\$30	\$36

Figure 3.3.13. The CAT4 diagrams above correspond to grouping *table rows of data for Client Members* and *table rows for totals*.

Similarly, to insert the ‘Max Balance’ function, we have inserted a new right category parent, also called ‘Totals’. This diagram emphasises the symmetry:

		Properties:	Properties:	TOTALS:
	Clients	Balance	Previous Balance	Max Balance
Members:	Mr. A	\$12	\$5	\$12
Members:	Mrs. B	\$24	\$48	\$48
Members:	Ms. C	\$54	\$54	\$54
TOTALS:	Sum	\$90	\$107	?
TOTALS:	Avg	\$30	\$36	?

Figure 3.3.14. The CAT4 representation is equivalent to this table layout, where the left parents of *Client rows* are now shown split in the left-most column, and right parents of *Client fields* are shown in the top row. RDB tables cannot have extra parenting rows or columns like this.

This is also essentially symmetric wr.t. *rows* and *columns*. The RDB model is not symmetric: it treats *rows as variables* and *columns as fixed*.¹⁰ Note also the cells intersecting to *row totals* and *column totals* are left unfilled: it is ambiguous from this table how they are calculated.

To specify the function, we first join the token: ‘*Max Balance*’, to an *instance of the MAX(x|y)* function (right), that we create for this purpose. Then we specify two parameter tokens: *Members* and *Balance*, connected by reference joins back to the data network. This means *functions are represented by network data, i.e. joins and fact tokens in the network, and hence they can be quantified over just like other data*. This allows high-order recursion (or quantification) very powerful. We explain functions in more detail in Part 4, but it is possible to define functions of all familiar kinds. The logic requires that we have a *recursive function base*, and are able to specify parameters to determine the fact tokens that the functions act upon, as well as the target (the tokens where data will be written to)¹¹.

Function code and interpreter.

We briefly discuss the CAT4 function code. This is defined further in Parts 4 and 5. The *function token* for a point *P* can store the *function string for P*, e.g. “=Sum([C])”, in its *Name* field. The general form is like:

- = Function([Source_Range],[Target_Range])

Or this may be put in a declarative form like:

- [Target_Range] = Function([Source_Range])

But this code is a secondary representation: the primary CAT4 representation is the *function token and its exterior points*. A *function interpreter* translates functions strings into the CAT4 graph.

- A *function interpreter* reads *function strings* and generates the *function tokens and parameter tokens* for the CAT4 representation of the function.

¹⁰ This is a limitation of RDBs and spreadsheets: although we can *transpose* tables or spreadsheets, both allow a limited numbers of *columns*, so we can transpose only limited numbers of rows into columns. CAT4 is symmetric w.r.t. transposition.

¹¹ CAT4 functions are more general than functions in spreadsheets or SQL table queries. In spreadsheets, functions must be entered in each cell individually: this can be done in CAT4 too, but not in SQL queries, where calculated fields are defined once in the column definition, which can also be done in CAT4.

This interpreter only applies if we use the function code to specify functions. But we may insert functions by inserting points directly.

The basic function code can be imagined as similar to functions in spreadsheet cells, and they can generally be written like: $=function(source ; target)$. To write *source* and *target* selections in CAT4, we can use the Boolean code: (A,B) , (A/C) , $((A,B)/C\dots)$, etc, where commas mean unions and bars mean intersections, and A , B , C , can be points, or further classes. These always resolve into finite ordered sets of points. We add the *EXT* and *INT* functions, the join and meet functions: $A \vee B$, $A \wedge B$, and the *relative position* functions, with $[x,y]$ vectors, and a $[CP]$ chain notation. Then we have a comprehensive logical reference system, for classes of *source points* and *target points*.

We also need functions to extract values from different content fields of the points, since they now have several fields. Consistent with normal conventions, we could use:

$[ID].NAME, [ID].VALUE, [ID].TIME1, [ID].TIME2, [ID].TV, [ID].ORDER$

to specify different content fields for the record of point $[ID]$. We might contract these to:

$.n, .v, .t1, .t2, .T, .O$ along with: $.id, .id1, .id2, .id3, .id4$

E.g. executing: $[104].v = Sum([102].v, [103].v)$ would write the *sum of values* of the two points 102 and 103, into the *Value* of point 104. We might just write: $[104].v = [102].v + [103].v$ in this case. But we can sum ranges, e.g. $[104].v = Sum([102] \vee [103].v)$. We can define functions to fetch data from classes of points, do calculations, and write results to appropriate classes of target points.

This raises an interesting point. The function code is in a *functional linear language*, meaning linear strings of symbols. Interpreting the function strings as CAT4 graphs is therefore a method for translating this kind of *language* into CAT4. Does this follow a general method? How does it relate to natural language, which also represents information using linear strings of symbols, i.e. words?

We will now briefly introduce the last main representational form, for treating natural language. We will see in Part 4 that the precise functional code language is just a special case of general language.

CAT4 propositional representation.

We earlier saw two related methods for representing information alternatively in the form of *fact complexes* or *propositional complexes*. Which method is best? How should they be used? They inter-translate through this fundamental transformation.

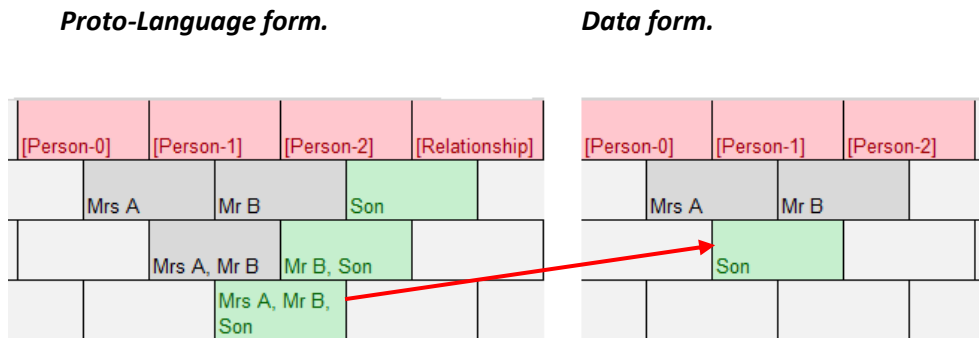


Figure 3.3.15. The same fact represented twice. Left is like a *propositional form of data*, right is a *fact complex* most suitable as a *data form*. *Mrs. A*, *Mr. B*, and *Son* are instances from categories, defined by the *Atomic Level* above them, which is singular, and collects all the lattices from below.

The *fact complex* form, on the right, is more compact for data storage, especially as data goes down several levels of detail. The *propositional form*, on the left, is like a *dimensional database*, with dimensions along the top row, values chosen in the second row, and hyper-dimensional cross-products populated below. It also resembles the linear form of a string of words or *phrase*. Note this works best when we have categories with finite discrete ranges, like *relationship types*. But when the value range is continuous, and characterised as a number, the *fact complex* is much better.

When we have numerical values or quantities in the facts, e.g. *Distance A to B is 3.14 meters*, we generally treat the numeric points as *tips*, or *primary value points* in the fact network.

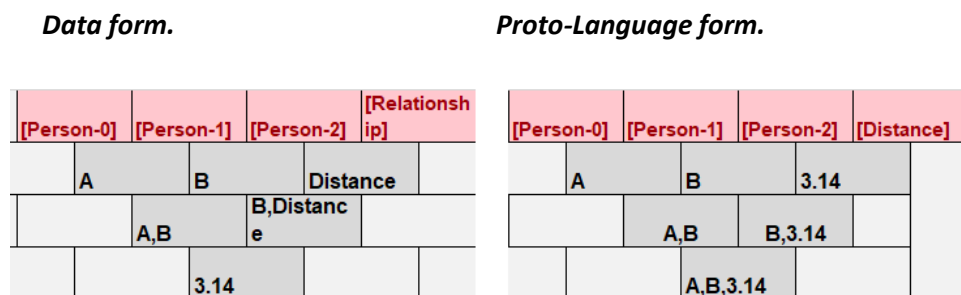


Figure 3.3.16. Left is the normal arrangement for storing numeric data, like “3.14”. Right resembles the construction of a proposition. Since values can vary continuously, we cannot make a complete list of possible values.

We normally *store information* in the form of fact complexes, as on the left above. We use other forms, more like *propositional complexes*, on the right, as convenient ways to represent or query the facts.

- This means we need to translate between these representational forms *within CAT4*.
- We can translate the “linearised” representations, as on the right above, into the fact complex data representations, as on the left.

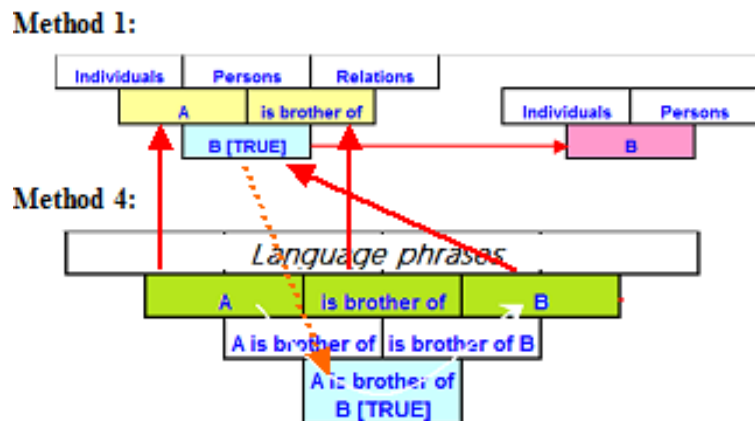


Figure 3.3.17. Two methods to represent the fact that *A is brother of B*. We translate Method 4 into Method 1, by inserting 3rd joins to identify the referents. Method 1 represents a fact complex directly, Method 4 represents facts indirectly, through a propositional complex.

The 3rd joins tells us what the Method 4 graph means, in terms of the *facts* in the main network. But note the two final *tips* (blue), that create the fact complexes, do not have 3rd joins. Instead, it is only the *truth values* that match between them. The *truth value* of the point: “B (TRUE)” above must match the *truth value* of the point: “A is brother of B (TRUE)”. But the first refers to the individual, *B*, as made clear by its 3rd join on the right, while we will take the second point to refer to *the proposition that: A is brother of B*. However, we still need a *function* to identify their truth values: this is not formally identified yet. Then we could use the second method to record a number of simple statements of this kind, and refer their truth to the fact network.

Taken literally in our semantics, the first point represents the first order proposition: *A is brother of B (is TRUE)*, from its three referents. The second represents: *the proposition that A is brother of B (is TRUE)*. This is identified from its interior construction, and the subsequent joins to “the facts”. However, we are now really using the joins in the “Language Phrases” graph for a special purpose, to construct propositions or phrases, with their meanings. We may now this as a special *function in*

CAT4. We have part of the CAT4 graph defined (with functions) as a *Language* partition, to represent language phrases, in a linear form similar to that above, and translate their *facts*.

Now we observe that the logical construction of points in Method 4 is quite automatic, and it can be generalised to construct phrases for any *string of points*.

CAT4 phrase tablet.

We call the following a CAT4 *phrase tablet*. We can generalise from combining just three terms, “A”, “is brother of”, “B” above, to any number of terms in a row. So we can apply this technique to analyse phrases or sentences.

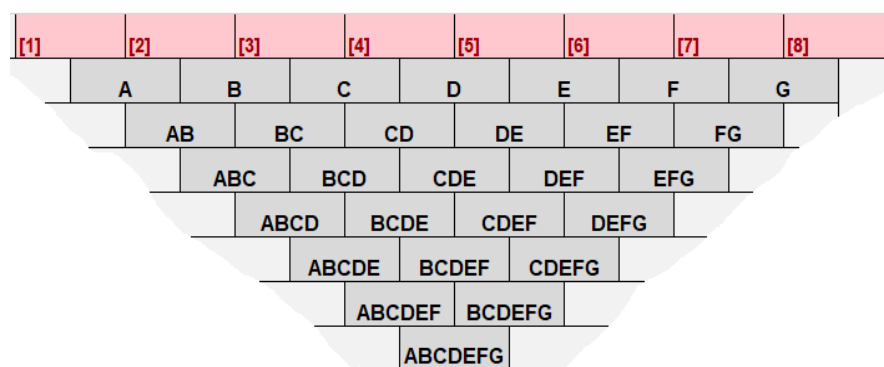


Figure 3.3.18. The CAT2 Tablet. All the neighbourly linear sub-sequences of a sequence: *ABCDEFGG*. The CAT2 interior of the point “*ABCDEFGG*” is the binary lattice, up to the sequence of individual symbols at the top.

This lets us associate a random bunch of points into one. We will then *interpret* the interior points, and as we will see, this will give us the means to associate the points in *any linear bracketed order*. And then we may allow some points to refer to *functions*, that tell us how to combine them.

However, for the simplest interpretation, we can use this as a fixed query on the fact network, looking for the *exterior intersection* of a sequence of *point values*, *A, B, ..., G*. So we associate the combined point: *ABC...G* with the function: $\forall(A,B,...,G)$. This gives *all points directly related to each point, A, B, C, ...G*. This can be done directly, or produced by the following recursive process.

- The *A, B, C, ...* are thought of as *parameters* for the query.
- We first associate each of *A, B, ..., G* with a *point* in the fact network, which is *the selection or interpretation of the parameters*.
- We then associate the points: *AB, BC, ..., FG* with the *pair-wise function, the intersection of exteriors*:

$$AB \rightarrow EXT(A)|EXT(B), \quad BC \rightarrow EXT(B)|EXT(C), \dots, \quad FG \rightarrow EXT(F)|EXT(G).$$

- And so on through the levels of the diagram, e.g. $ABC \rightarrow EXT(A)|EXT(B)|EXT(C)$, etc.
- This works algebraically because intersection is associative. (So: $ABC \rightarrow EXT(A)|(EXT(B)|EXT(C))$ is equivalent to: $ABC \rightarrow (EXT(A)|EXT(B))|EXT(C)$.)
- At the bottom we get a single fact that is related to *all the points*, and represents the intersection of all exteriors: $ABCDEFG \rightarrow EXT(A)|EXT(B)|\dots|EXT(G)$.
- This is the function: $\forall(A,B,\dots,G)$.

Now this is like a method for forming prototype sentences, but it is limited in this example to a single query function, viz. *intersections of exteriors*. We may also define a similar function for *unions of exteriors*, e.g. to combine all information about both A and B. And of course in general, we want more complex queries, e.g. to find all information about *either* A or B that involves C, etc. So we will *generalise the function*, from this simple fixed function.

We will also make it functionally recursive, so different ‘query functions’ are specified in the A, B, C.. parameters themselves.

We also need to generalise the method of *reference*, because a combination of terms may give multiple points in the *Fact* network, but the symbols in the tablet have only one 3rd join each. We will assign referents by adding individual points double-joined below the symbol points. We illustrate some of the concepts with an example.

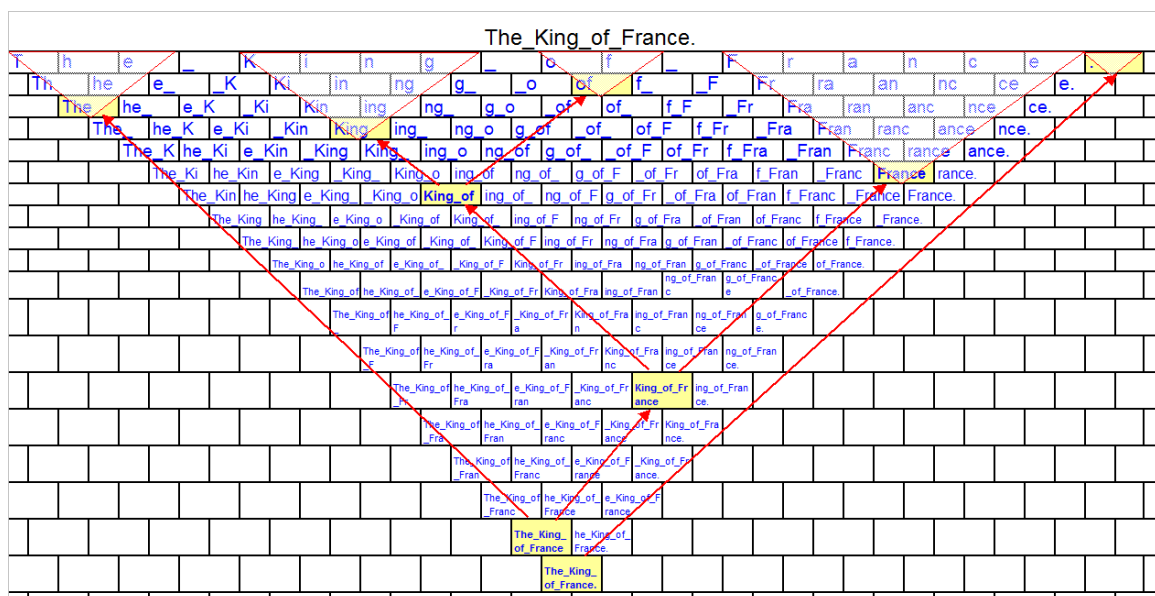


Figure 3.3.19. A *tablet* used to analyse a phrase, constructing it from the individual letters up. We pick out meaningful symbols that are produced, and assign them *interpretations in the network*. This generates a *binary tree* over the CAT2 tablet.

- Proposition. The background CAT2 tablet represents all possible ways to combine a linear set of symbols by bracketing them with each other. [Proof: exercise].

We have chosen to bracket the symbols as follows, first bracketing the initial words from the letters, then combining the words in this order: ((The ((King of) France)).) Other bracketings are possible, and may make sense, and may construct the same final proposition, or (for ambiguous phrases), different propositions. What determines this *binary phrasing*? There are two kinds of answers:

- Ideally: the meaning we want the phrase to be interpreted with – Semantic Accuracy.
- Practically: the previous examples the software has to refer to – Machine Learning.

Semantic accuracy is the primary goal. But in a software implementation, machine-learning algorithms will determine most interpretations, aiming to conform to previous use. (Unless overruled by a user teaching the machine to adopt different meanings.) But regardless of implementation, the goal is *semantic accuracy*.

Note what we do here is make a *semantic construction in parallel with a syntactic construction*. We start with the *complete symbol*, the phrase: “The King of France.” Then we deconstruct it into its linear string of symbols. Then we join them up again in an iterative *construction*, which ends up with the complete symbol again. The special thing is that, as we join them up to form the *binary tree*, we require each node to be interpreted *semantically*. They should be able to be *assigned referents to facts or functions* represented in the general *fact network*. And these *referents* should combine semantically in parallel with the syntactic phrase construction.

- Symbolic strings can be atomised then recombined recursively in the CAT4 phrase tablet, to form a hierarchy of *functional terms*, in the process of constructing the phrase or sentence.
- In a symbolic phrase, first of all there are usually several ways to combine symbols functionally. So we scan it for familiar words or phrases, and fix an interpretation as a *binary graph*.

Phrase compaction and referents.

We can *compact* these graphs, simplifying symbolic clusters, and we also introduce *reference tokens* for phrases, as a formal way of specifying the interpretation.

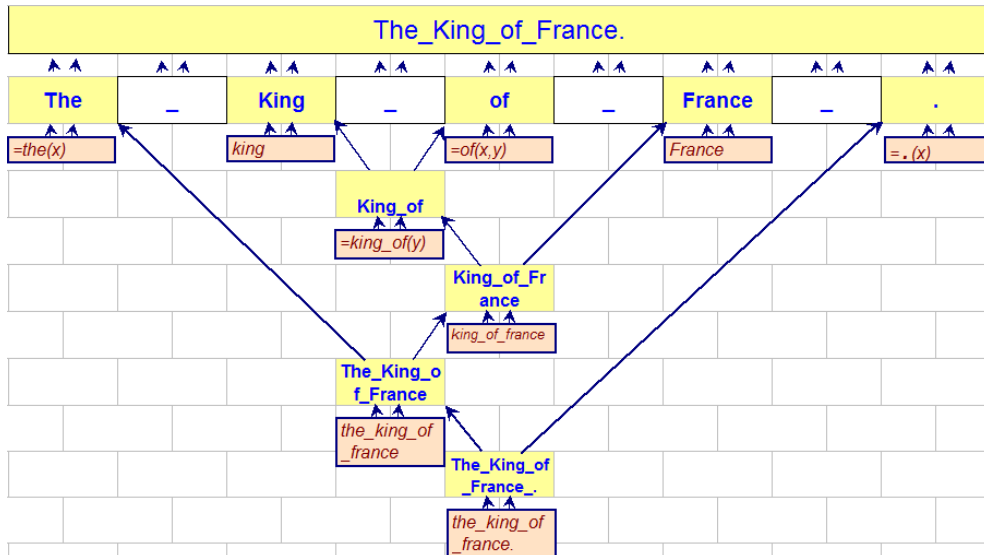


Figure 3.3.20. Compacted version of the phrase tablet, with points for *reference tokens*. The latter are joined the fact network, rather than the phrase symbols directly.

When we populate it with joins to the fact network, we have both ordinary facts and functions to refer to.

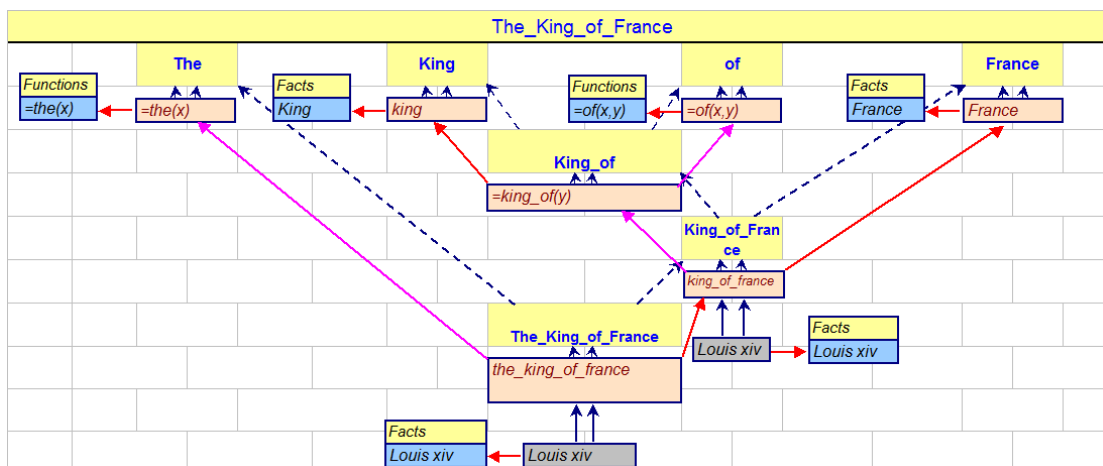


Figure 3.3.21. A phrase tablet populated with the *fact interpretation*. This represents the semantic relations of the phrase, in reference to the CAT4 facts. Note the referent points refer directly (third joins) to points in the fact network. Note the *semantic interpretation* is represented by the joins; the *extensional content* is produced by executing the functions.

At a certain time in history, this phrase would have referred to Louis XIV, and the information shown above would indeed have been correct. But of course, no more. For detail of a point, consider the referent of *King of France*, which historically has more members.

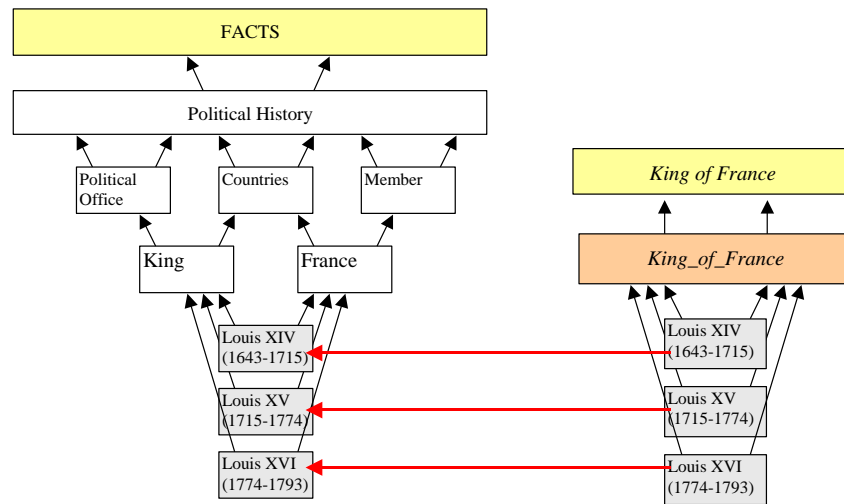


Figure 3.3.22. Joining *phrases* to *facts*. On the left is some information in the fact network, on the right is a detail of the *phrase*, “King of France” in the phrase network. When the referents form a *class* they are stored (grey) below the immediate reference point (apricot), which is below the phrase point (tan).

Note double-joined points below a CP (collection point) mean *class content of the CP point*. Here, “King of France” is the collection point for a class of three *referent points*.

The individual *node points* in the phrase network have this join structure.

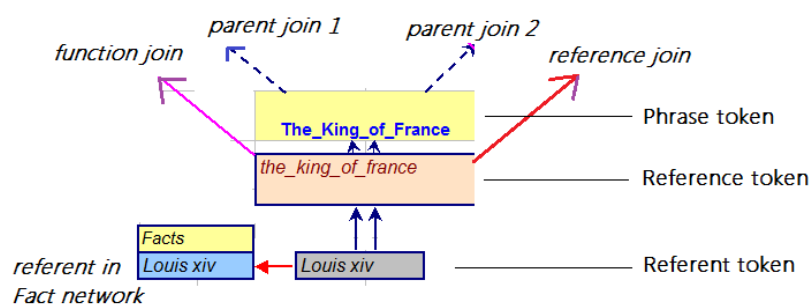


Figure 3.3.23. Phrase points the tablet construction.

Now we can reconstruct the earlier phrase diagram, for this case when we have multiple historical *kings of France* to reference (and there is no *present King of France*).

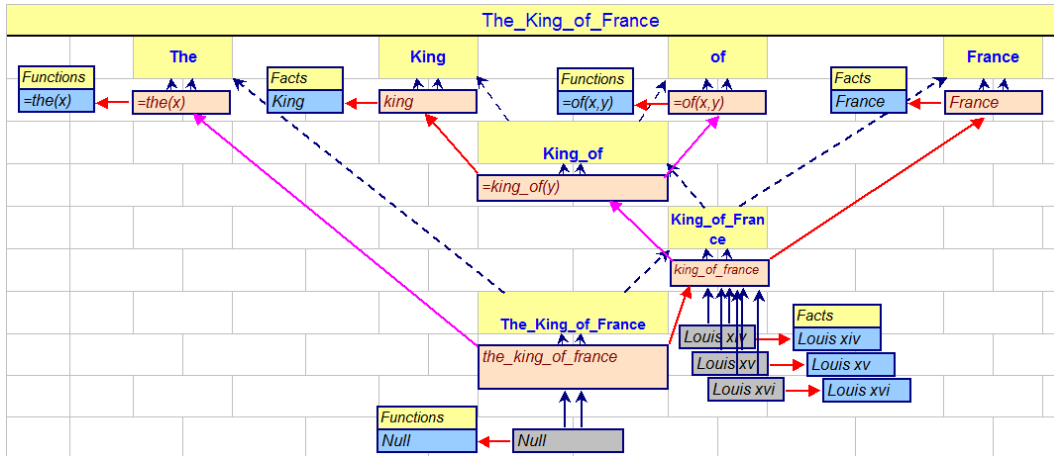


Figure 3.3.24. This illustrates the extension of *the King of France* if we have multiple referents for “*King of France*” in the fact data. (Here we show three.) This means there is no unique value for the *King of France*. So the final referent for the phrase is *null*.

Note we use a special token here to explicitly represent *null*. If we simply left the extension blank, we could not tell if the calculation had been executed. This brings the role of the term: “*the*” into play. This has the role of a *function*, applied to a descriptor. In this case, we identify several individuals as being *King of France*, and when we apply the function labelled: “*=the(X)*” to this, it returns *null*, because there is no unique member. Hence there are no referents for “the *King of France*”. Note the function does not return *False*. It returns – nothing, an empty class. But we populate the reference point with a *Null* point, which is a special logical point, used to specifically record that classes are empty.

Note we now have a unified account of linguistic constructions, which is not based around propositions, but treats all phrases equally. A propositional statement is simply one that resolves to a *truth value as its referent* (i.e. extension). But we use *exactly the same formal method as above* to calculate the referent of a *propositional statement* as we do for any other phrase.

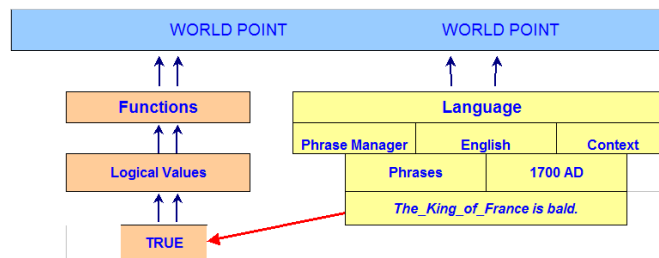


Figure 3.3.25. A *propositional statement* is a phrase that evaluates to *True* or *False* or *Null*. Its formal analysis in the phrase tablet is continuous with its component phrases.

CAT4 Natural Language Translator.

Obviously this system needs more formal discussion, and we explain more details in Part 4. But we conclude by summarising some key features of the solution here. The NL translator is similar to RDB and GraphDB translators in structure:

<u>External Source:</u>	<u>CAT4 Transform</u>	<u>CAT4 Data</u>
Tables →	[CAT4 Table] →	CAT4 Facts
Graphs →	[CAT4 Graph] →	CAT4 Facts
Phrases →	[CAT4 Phrases] →	CAT4 Facts

The intermediate ‘transforms’ are the representations, respectively, in the CAT4 *phrase tablets*, *table graphs*, and *bi-graph graphs*, as we have seen. Their main feature is that they are clear structural *transcriptions* of the *symbolic information* into CAT4 graphs. Phrases are the simplest in this sense: *there is only a string of symbols*. But once we have brought them into this “CAT4 Transform” representation, we must *interpret them*, by associating them with the rest of the CAT4 fact graph. This is most easy for the structured data formats (tables, bi-graphs), and most difficult for the unstructured (natural language).

However, CAT4 offers a special advantage for a NL processing model, because of the coincidence between the *CAT2 Lattice*, and the *Binary Phrase Tree*. It is possible for this to work as the representational model within CAT4, because CAT4 is itself *semantically complete*. Hence the *semantic model* required to relate NL phrases to facts is already encapsulated in the CAT4 semantic system.

The part of the model that needs automation is then the processing of phrases into interpreted phrases. As this proceeds, the results of previous interpretations provide the guide to new ones. This is the most basic Machine Learning function. Since it is possible to recursively iterate functions, including machine learning functions, the system can learn interpretational or grammatical rules. In fact, this is how it represents such rules: *by applying algorithms to select referents using previous examples as templates*. One may adjust the algorithms and example-sets too, these are dynamic parts of the learning process too.

This *NL Translator* may be compared to Noel Chomsky’s idea of a *Universal Grammar*, a capability for learning natural languages that we as humans are born with innately. This means for Chomsky some kind of generalised ‘algorithm’ for representing and processing *natural language grammars*. The evidence is that we do learn languages, as babies, children, and as adults if we want. We learn

the grammatical rules of new languages. All people can learn all kinds of languages. So there is some common “Universal Grammar” or meta-algorithm behind all these specific *instances* of grammars and languages.

The CAT4 NL Translator provides a design very well suited to this. However it is not a rule-based “grammar” or digital logic circuit quite like either Chomsky or the linguistic-logicians or the programmers expect. It is a Machine Learning system. The “grammars” of our languages are kept in the records of previous language use. (Remember we learn grammars also by getting linguistic instructions). We use these records for the rules to identify grammatical parts of sentences, and match interpretations. We use them to recall facts. They provide a network of joins that can shape facts, and they act as facts themselves. Hence what we regard as *grammatical rules* are not logical rules, they are malleable and plastic.

At this point, we have come in an odd circle, arriving at something like Wittgenstein’s later conclusion, referred to as: “meaning is use”. The CAT3 network is reminiscent of Wittgenstein’s early vision in the *Tractatus*, of a world of facts, in logical order. This seemed to mean language should be ‘logical’ too. But when we examine the reality of language, it is always more or less blurry, and there is no guarantee that the facts we think we talk about are the same facts, with the same meaning, for all of us. So the later Wittgenstein concluded the early logical theory is not applicable as a model or philosophy of real language. He turned away from the very concept of *meaning*, and wanted to replace it with *use*. However we believe both are real parts of the model of language.

To conclude, we compare CAT4 with an advanced formal semantic theory, of which Wittgenstein was not aware.

Appendix 3.3.1. Semantic completeness and comparison with TIL.

We need to consider whether our system is *semantically complete*. After starting with CAT2, we had to add *Time* and *Truth* to the fact-level, making them intrinsic elements of our fact model, to capture two crucial aspects of facts and their propositions (they apply to time intervals, and they may be true or false.) We required the 3rd *reference join* and the 4th *function join* in addition to the first two *joins*. How do we know if there are more aspects we have not recognised? We can check.

(i) There are practical checks, showing how to translate common forms of information into CAT4, and (ii) There are theoretical checks, comparing with other semantic systems.

Here we propose, for (ii), to compare CAT4 with a leading system of formal semantic logic, TIL. There are other systems of logic, but TIL is special, in being probably the most *complete*. It is called a *hyper-intensional logic* [Duzi *et al*, 2010]. Note first some basic pre-intensional concepts.

- Ordinary formal logic takes *propositions* as primary entities embodying information, and characterises propositions as *true* or *false*.
- Boolean logic treats propositional connectives, as simple relations on a class of *propositions* and a class of *truth values*. But this gives us no model of the internal construction of propositions.
- In language, we state propositions as complex relations, and they are logically related to each other by their *constructions*.
- The predicate calculus (Frege) analyses the internal construction of propositions in terms of (complexes of): *individuals, relations, variables, quantifiers*.
- E.g. *There is something red* may have its logical construction translated: $(\exists x)Red(x)$. Logical relations are represented by functional application and by quantification.

However the predicate calculus proved inadequate for the logic of empirical language, and logicians had failed by the 1930's to provide a semantic logic suitable for analysing natural language. This is because of a failure to represent possibility, contingency and other aspects of meaning in extensional logics. Much of our ordinary and scientific inferences are about possibilities, and much of our reasoning is counterfactual, involving reference to propositions that are *undecided or actually false*. "If I don't do X, event Y will happen." Suppose this is true, but I do X. Then it is about something that never happens. Yet it is true. I would also say, before the chance to do X elapses: "It is possible for me to X, or not to. If not, Y will occur." Hence the proposition X may be possible but not actual, and propositions become actually true or false according to the contingent passage of time.

Propositions have logical relations independent of their actual truth. We must therefore have a way of reasoning with actuality distinct from possibility. In the 1950s-60s, logicians began to formally introduce *possible worlds* into a logical calculus, to model the logical features of propositions and natural language in a more complete way, including modal propositions and intensional contexts. This led to modern *semantic logics*. Montague [1970/1] and Tichý [1971] were the first to publish systems of *possible-world intensional logics*. Tichý's theory culminated in *Transparent Intensional Logic*, TIL, [1987]. This includes both *intensions over possible world-times*, and *constructions*, a further fundamental extension introduced by Tichý to make the system *semantically complete*.

Other logical systems have points where they are incomplete for the representation of natural language semantics.

TIL is called a *hyperintensional logic*. [Duzi, et alia 2009]. Logicians in this area since Tichy's [1987] have done considerable work, showing how natural language meaning can be represented through this logical formalism. As a *formal semantic logic and theory*, TIL provides the most advanced theoretical comparison for CAT4.

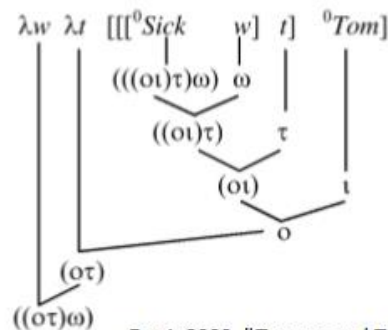
- We can try to match concepts directly between CAT4 and TIL ontologies.
- We can use TIL, as a semantic meta-theory, to analyse CAT4 semantics.
- We can use CAT4, as an information system, to analyse TIL semantics.

Here we briefly discuss the first and second points. We want to see how the two systems match up, and especially if there is any noticeable omission from CAT4. The third step comes when we continue with the method for translating *linear languages* into CAT4, in Part 4.

$$\lambda w \lambda t [{}^0 Sick_{wt} {}^0 Tom].$$

This construction is the literal analysis of the sentence "Tom is sick".

Ad (3) Type-theoretical checking:



Duzi, 2009, "Tenses and Truth Conditions", p.6

Figure 3.3.26. This is an example from Mari Duzi, illustrating a TIL analysis of a simple tensed expression: "Tom is sick."

TIL provides a formal logical language, in which to represent the semantic structure of NL phrases. This is illustrated above, in a "logical translation" of a sentence. CAT4 also provides a logical translation of the same NL phrases, so we can look at how they match up. Although they look completely different at first, at a fundamental level we will see they have very similar logical bases. In terms of translating NL, TIL is quite successful at defining a functional base for interpreting semantics. However, TIL formulae, like natural language sentences, are no good for *storing*

information. And there is no automated method for translation or analysis of NL into TIL. Indeed, it is a tedious and difficult task. A practical TIL system must refer to another method for primary information storage, if it is to perform a task like CAT4, of recording and evaluating NL sentences, and comparing them against other recorded data. In CAT4, we find that the *data storage* i.e. the *representation system*, is the fundamental key, and as TIL lacks this, it means TIL cannot represent certain things that *require the data system and queries*, such as grammatical rules stored in data.

However, in the detailed application to a single instance, we should hope CAT4 and TIL have similar semantic analyses, and the same kind of expressive power. We note below that is true.

As an example of TIL script, the meaning of “Tom is sick” is represented by the construction: $\lambda w \lambda t [{}^0 Sick_{wt} {}^0 Tom]$, in the symbolic language of TIL. This *proposition* is interpreted as a mapping from worlds and times to truth values, and may be applied to *the actual world, now*, to give a truth value (or null). Below the linear expression in the diagram, we see a diagram of *type theoretic checking*. This appears similar to the network diagrams of CAT4 used to construct the meaning of sentences. Note this appears as a simple *two dimensional graph (flat lattice)*, which interprets the ‘functional composition’ of the linear code. Functional code in maths and logic, with nested argument structures, corresponds to such two dimensional ‘lattices’ when we specify all the internal relations between parts of the code, as we do in a *graph*. So beneath the linear surface of the TIL code are hierarchical structures of functions, intuitively similar to the CAT2 networks.

If we can match the CAT4 concepts to those of TIL, we can get an immediate comparison for semantic completeness. TIL constructs its entities from a formal base, using a class of *domains*, which initially includes four classes of objects. These are essential logical classes required for semantics in this view: *individuals, I; worlds, W; times/real numbers, T; truth values, O*. TIL also has *constructions, χ* , which includes variables and complex functional objects, such as *properties, offices, propositions*, etc. But ignoring constructions for the moment, the first order *base* provides the key ‘empirical objects’ to represent simple factual propositions.¹²

¹² Note this *construction over a base* appears similar to how entities of classical physics (like energy, velocity, momentum, etc) are modelled from base classes of *individuals; time; space; mass; charge*. In this case, the special classes defining *worlds* are classes of 5-tuples defining space-time trajectories: $r(i,t) \equiv \{(i,t;r,m,q)\}$. The condition for classical mechanics is that r is a continuous differentiable function of t . How close is this analogy?

Marie Duží gives a most succinct definition:

“There are two kinds of constructions, atomic and compound (molecular). Atomic constructions (Variables and Trivializations) do not contain any other constituent but themselves; they specify objects (of any type) on which compound constructions operate. The variables x, y, p, q, \dots , construct objects dependently on a valuation; they v -construct. The Trivialisation of an object X (of any type, even a construction), in symbols $0X$, constructs simply X without the mediation of any other construction. Compound constructions, which consist of other constituents as well, are Composition and Closure. The Composition $[F A_1 \dots A_n]$ is the operation of functional application. It v -constructs the value of the function f (valuation-, or v -, -constructed by F) at a tuple argument A (v -constructed by A_1, \dots, A_n), if the function f is defined at A , otherwise the Composition is v -improper, i.e., it fails to v -construct anything.⁵ The Closure $[\lambda x_1 \dots x_n F]$ spells out the instruction to v -construct a function by abstracting over the values of the variables x_1, \dots, x_n in the ordinary manner of the λ -calculus.⁶ Finally, higher-order constructions can be used twice over as constituents of composite constructions. This is achieved by a fifth construction called Double Execution, $2X$, that behaves as follows: If X v -constructs a construction Y , and Y v -constructs an entity Z , then $2X$ v -constructs Z ; otherwise $2X$ is v -improper, failing as it does to v -construct anything. TIL constructions, as well as the entities they construct, all receive a type. The formal ontology of TIL is bi-dimensional; one dimension is made up of constructions, the other dimension encompasses non-constructions. On the ground level of the type hierarchy, there are non-constructional entities unstructured from the algorithmic point of view belonging to a type of order 1. Given a so-called epistemic (or objectual) base of atomic types (o-truth values, ι -individuals, τ -time moments/real numbers, ω -possible worlds), the induction rule for forming functional types is applied: where $\alpha, \beta_1, \dots, \beta_n$ are types of order 1, the set of partial mappings from $\beta_1 \times \dots \times \beta_n$ to α , denoted ‘ $\alpha \beta_1 \dots \beta_n$ ’, is a type of order 1 as well. Constructions that construct entities of order 1 are constructions of order 1. They belong to a type of order 2, denoted ‘ $\star 1$ ’. The type $\star 1$ together with atomic types of order 1 serve as a base for the induction rule: any collection of partial mappings, type $(\alpha \beta_1 \dots \beta_n)$, involving $\star 1$ in their domain or range is a type of order 2. Constructions belonging to a type $\star 2$ that v -construct entities of order 1 or 2, and partial mappings involving such constructions, belong to a type of order 3. And so on ad infinitum.” Duží, 2009, p.4.

The TIL ontology corresponds very closely to the CAT4 *ontology of facts*.

INDIVIDUALS. CAT4 meta-theory assumes there are *referents* for fact-tokens, which are entities in the world (as well as logical entities and functions). This *external reference* is not defined *within the CAT4 graph*. External referents or individuals are assumed in the interpretation.

- The CAT4 Fact Table is the complete list of all *individuals* presently referred to.
- Each CAT4 Fact Table provides a finite list of such *individuals*, but there is no complete list of *all individuals in the world*.
- The 3rd join represents *referential identity* explicitly.
- The class of *primary points* in CAT4 represents an ontology of individuals, with a unique fact-token for each individual.

- Each fact is also an object of reference in its own right. It is named with a unique ID number, and may be referred to in functions or *constructions*.
- Each fact-token constructs an atomic proposition through *its own referent and the referents of its two immediate parents*. These correspond to first-order propositions of TIL.
- Many distinct *facts* in CAT4 may correspond to the same first-order proposition.

TIME. Each fact in CAT4 is associated with a *time interval*, representing *time* similarly to TIL.

However CAT4 has an *interval semantics for facts*, while TIL has point-time semantics. Facts at a specific moment t are represented in CAT4 at the interval: $[t, t]$. However CAT4 cannot represent an infinite number of distinct momentary facts.

TRUTH. Each fact-token has a truth value, determining its *first order proposition about the fact*. This is generally not the *actual truth value*, i.e. the *material truth*, unless it happens to represent a true fact in the real world.

- If we assign a token P the value *False*, we change its *first order proposition* from P to: $\sim P$. It refers to the same *fact*, but now represents it as not actual.

WORLDS. The fourth domain class in TIL is *Worlds*. TIL *intensionalises* on worlds. Intensionalisation in CAT4 is represented by the *fact interior*, and all the CP's up to the Zero Point, which we (normally) interpret as referring to the actual world. The interior is structured, with Collection Points in a hierarchical structure, and each CP can modify the *referential meaning* of its exterior points.

CONSTRUCTIONS. The most unique feature of TIL, *constructions*, corresponds to execution of functions over the CAT3 lattice, and recursive execution of *function joins* in CAT4.

- These joins represented in the Fact table ensure there is first order quantification over constructions.
- Trivialisation corresponds to obtaining the *primary reference point*.
- Application of functions to arguments corresponds to: $\forall(P, Q, \dots, R)$.
- Abstraction is provided by ordinary recursive functions, inserted with the 4th join.
- Execution corresponds to executing functions, though the 4th join trees.
- Double execution corresponds to executing functions recursively.

TIL has classes of *worlds*, and *world intensions*, while we have not mentioned these in CAT4. But CAT4 does identify the Actual World as the normal referent of the Zero Point. Identifying the

corresponding role of *worlds* or *world intensions* in CAT4 is a puzzling point, but since the Zero Point naturally and intuitively refers to The World, this must correspond to *world-intensionality* in TIL.

Instead of *world references*, CAT2 has the general parenting structure, *ID1, ID2*. The parents give meaning to the fact-tokens, so a given token represents a *fact* (a complex of entities) and is not just an *object reference* (an isolated symbol). The whole *interior* of a CAT2 fact is ultimately involved in its meaning, up to the Zero Point. The Zero Point is the *Primary World Reference*. We take the Zero Point to refer to the external reality or actual world.

- The significance of a single Zero point is that it means there is a *single world-reference*, or a single, maximal truth-determiner.

Note this is a metaphysical assumption behind ordinary logic: that there is a single, maximal Truth. This metaphysics allows the Truth to *change with time*, but assumes there is a unique complete truth at every moment time.

If we want to contemplate *multiple actual worlds*, as some philosophers do, in CAT4 we might allow multiple “zero-points” (or roots) in a CAT4 relation, with some contradictory facts. But this makes functions unstable. More simply, we can represent facts characterising two distinct Worlds much as we represent distinct points of view: with two distinct CAT4 relations, or with *intensionalised representations*. Within CAT2 we have *Collection Points*, which serve, by semantic convention, to modify or determine the *epistemic or metaphysical status of their exterior points*. This is similar to *world-intensions* because we can quantify over these points. This can modify meanings up a chain of CP’s.

Appendix 3.3.2. CAT4 top-down metaphysical categories.

In CAT4 graphs of our larger belief systems, immediately below the *Zero Point* we have a small set of highest-level categories, CP’s. There are six we will call: *Facts, Fictions, Functions, Data Sets, Language, Agents*.

- In *Facts* we put our records of the actual facts (as we believe them to be).
- In *Fictions* we may put all kinds of facts (true or false), even about fictional characters and entities that *do not exist in Facts*. These have only references within *Fictions*.
- In *Functions* we represent the available functions, and add points in their exteriors to represent parameters and instances.

- In *Data Sets* we store imported information, under suitable categories to identify the origins. There is no implication of the actual truth of the information, e.g. in imported CAT4 records from other agents. These records represent the information state of external CAT4 or other systems.
- In *Language* we store conversations, as sequences of phrases, and their interpretations in the *phrase tablets* specifying reference to *Facts, Functions, Fictions etc* partitions.
- In *Agents* we represent other agent's beliefs. This allows intensional belief contexts. When Agent's beliefs are represented in CAT4, they may have recursive beliefs about us.

Several further partitions come up in practice. These categories appear like metaphysical divisions, at the highest levels of our conceptual and cognitive organisation. In particular, we cannot reason objectively without defining our beliefs about what is *actually true*, and separating this class from propositions we may represent as *counterfactual, false or fictional*. We will discuss this example.

Fictions are generally speaking counterfactuals with non-actual entities like persons, places, etc, but they are not just falsehoods. Dealing with fictions is a surprisingly difficult task in most semantic systems, raising many disputes about abstract entities and the like. We may characterise *fictions* as *not true of the actual world*, but they are not just *false*. In CAT4, fictional facts are simply *not about the actual world*.

Note that some facts represented in fictions are actually true, e.g. Napoleon was at Borodino and Moscow in reality, as in the fictional story *War and Peace*. Other facts are what we call *true in the fictional context*, e.g. Pierre and Andrei were together in Moscow at certain times, but only in the story. But who are *Pierre and Andrei*? What external entities do these facts refer to? The answer of course is: no one actual. They refer only within the *world of the fiction*.

- In CAT4 we partition *Fictions* at the very top, separately from *Facts*.
- This gives the entire content of *Fictions* a different *intensional interpretation*.
- Fictional status is reflected formally by the 3rd join reference chains of the fictional entities, which remain entirely within the *Fictions* partition, and have fictional primary points.
- Whereas real entities mentioned in *Fictions* have 3rd joins to primary points in *Facts*.

So in CAT4, we make a partition called *Fictions*, and this has a meaning for us, which we apply through functions, which may treat different partitions differently. In terms of *its intension*, it parents to the *Zero Point*, which refers to the actual world. Thus it is intended to mean *The Actual Facts about Fictions*. Similarly content in *Functions, Agents, Data Sets*, etc.

Thus these primary partitions under the Zero Point represent facts of distinct *epistemic character*. These partitions are required for the formal network organisation if nothing else. They separate functionally similar groups. But how fundamental are they? And do we need to base our representational system around the special character of different “metaphysical types”?

Do they represent *a priori* necessary categories of thought? Or are they subjective quirks of our conceptual organisation? We could certainly lump them together differently, e.g. put *Agents* under *Data Sets*, or *Fictions* under *Facts*, but they would still have to retain their own intensional sub-categories. Because we will treat them differently functionally, according to their *intensional category*.

- The intensional categories are identified by the *Collection Points* in the interior chains of the primary reference points.

But different people may want to use different divisions. Even our category of “*Actual Facts*” might be divided, e.g. into: *Bible Facts*, *Science Facts*, *Legal Facts*, *Opinion Facts*, reflecting some attitudes towards different types of authorities. We could have functions to make the authority of one override another, if there was a difference of opinion. I don’t think this is a good idea, but we could do this if we wanted. Our minds do not all have the same logical organisation of concepts, and we do not all think in the same ways.

We can represent such variations of belief, as reorganisations of *content and category systems* in CAT4. However some distinctions, like *Facts and Fictions*, appear quite fundamental, and must be reflected in functions. In implementing CAT4, with its top-down architecture, we suddenly find ourselves in the realm of Kantian metaphysics, intensional logics about reflexive beliefs, and modal contexts. Representing beliefs in CAT4 exposes the rational organisation of our higher-level beliefs.

In terms of the framework for *factual information* however, e.g. if we think of business information, facts about persons, products, accounts, etc, everyone is expected to have similar CAT4 representations of first order propositions, or first order tables. When we get down to the detailed data, representation is quite concrete.

There may be differing opinions on how to set up the higher ‘metaphysical categories’ of a CAT4 system, but the important thing for now is that the *epistemic-metaphysical divisions broadly make sense to us for representing factual and non-factual propositions*. Consider how our semantics work for *Fictions*.

The intuitive meaning of *Fictions* is that some referred entities are ‘invented’, and have no external object reference outside their fictional context. With facts that refer to real-world entities (e.g. Napoleon, Moscow), their fact-tokens in *Fictions* partition refer, via the 3rd join, to *primary facts* in the *Facts* partition. But entities like “Pierre” are only referred to within *Fictions*. Their primary referent ends in the *Fictions* partition. There is a class of entities that have their primary referents ending in *Fictions*, and is even further partitioned into distinct works of fiction, which are CP within *Fictions*. This is the *fictional ontology*.

As long as there is a common understanding of what is fictional, the condition that *fictional entities do not refer to the external world* lets us formally identify *fictions*.

- To start using the system, we have to begin top-down, by imposing some high-level *metaphysical divisions of facts*. These are divisions that appear to have a kind of universal logical function to us.
- The Categories below the Zero point, or CP’s generally, do not modify the meaning of propositions in their exteriors, but they modify the intensional assumption, i.e. the relation to the world (actuality).

References.

- Bridges, Jane. 1977. *Model Theory*. Clarendon Press.
- Carnap, Rudolf, 1947, *Meaning and Necessity*. University of Chicago Press.
- Chang, C.C. and H.J. Keisler. 1973. *Model Theory*. North-Holland.
- Church, Alonzo. 1956. *An Introduction to Mathematical Logic I*. Princeton.
- Codd, Edgar Frank (June 1970). "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM*. **13** (6): 377–387. doi:10.1145/362384.362685. S2CID 207549016.
- Date, Chris. 2004. (8th Ed.) *An Introduction to Database Systems*. ISBN 0-321-19784-4
- Durbin, John R., 1992. *Modern Algebra: An Introduction*. Wiley and Sons.
- Duží, Marie. "Intensional Logic and the Irreducible Contrast between De dicto and De re", <http://www.cs.vsb.cz/Duží/>
- Duží, M., Jespersen, B., Materna, P. 2010. *Procedural Semantics for Hyperintensional Logic: Foundations and Applications of Transparent Intensional Logic*. Springer Verlag.
- Frege, Gottlob, 1879. Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens. Halle a. S.: Louis Nebert. Translation: Concept Script, a formal language of pure thought modelled upon that of arithmetic, by S. Bauer-Mengelberg in Jean Van Heijenoort, ed., 1967. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press.
- Frege, Gottlob, 1884. *Die Grundlagen der Arithmetik: Eine logisch-mathematische Untersuchung über den Begriff der Zahl*. Breslau: W. Koenner. Translation: J. L. Austin, 1974. *The Foundations of Arithmetic: A Logico-Mathematical Enquiry into the Concept of Number*, 2nd ed. Blackwell
- Frege, Gottlob, 1892. "On Concept and Object." (First published in the *Vierteljahrsschrift für wissenschaftliche Philosophie*, 16 (1892).
- Holster, Andrew T. 2008-2011. "System and method for representing, organizing, storing and retrieving information." US. Patent Number: 7,979,449. July 12, 2011.
- Holster, Andrew T. 2021 (a). "Introduction to CAT4. Part 1. Axioms." [Introduction to CAT4: Part 1. Axioms, viXra.org e-Print archive, viXra:2101.0089](#)
- Holster, Andrew T. 2021 (b). "Introduction to CAT4. Part 2. CAT2." [Introduction to Cat4: Part 2. Cat2, viXra.org e-Print archive, viXra:2101.0088](#)
- Materna, P. 2004. *Conceptual Systems*. Berlin: Logos.

- Materna, Pavel. 1998. *Concepts and Objects*. Acta Philosophica Fenica, vol. 63, 1998.
- Montague, R. 1970. "Universal Grammar". *Theoria* **36**, pp. 373-398.
- Montague, Richard. 1973. "The Proper Treatment of Quantification in Ordinary English". *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*. D.Reidel.
- Robinson, Ian, Webber, Jim and Eifrem, Emil. 2015. *Graph Databases (2nd Edition)*. NEO4J.
- Russell, Bertrand. 1905. "On Denoting", *Mind*, Vol. 14. ISSN 0026-4423. Basil Blackwell.
- Tarski, Alfred. 1983 (1956). *Logic, Semantics, Metamathematics: Papers from 1923 to 1938 by Alfred Tarski*, Corcoran, J., ed. Hackett. 1st edition edited and translated by J. H. Woodger, Oxford Uni. Press.
- Tarski, Alfred. 1933 "The Concept of Truth in Formalized Languages" and 1936 "On the Concept of Logical Consequence", in [1983].
- Tichý, Pavel. 1971. "An Approach to Intensional analysis", *Nous* **5**, pp. 273-297.
- Tichý, P. 1988. *The Foundations of Frege's Logic*. Walter de Gruyter.
- Tichý, P. 2004. *Pavel Tichý's Collected Papers in Logic and Philosophy*. Svoboda, V., Jespersen, B., Cheyne, C. (eds.), Dunedin: University of Otago Publisher, Prague: Filosofia.
- TIL (Transparent Intensional Logic) Website: <http://www.phil.muni.cz/fil/logika/til/index.html>
- van Benthem, Johan and Alice ter Meulen. 1997. *Handbook of Logic and Language*. M.I.T. Press.
- Wittgenstein, Ludwig. *Philosophical Investigations*, 1953, G.E.M. Anscombe and R. Rhees (eds.), G.E.M. Anscombe (trans.), Oxford: Blackwell.
- Wittgenstein, Ludwig. *Tractatus Logico-Philosophicus (TLP)*, 1922, C. K. Ogden (trans.), London: Routledge & Kegan Paul. Originally published as "Logisch-Philosophische Abhandlung", in *Annalen der Naturphilosophische*, XIV (3/4), 1921.
- Wikipedia pages.

Acknowledgements.

Special thanks go to Jeremy Praat for early discussions of the CAT2-CAT3 concept. Thanks to Patricia Holster, Bev & Gary Thomson, Peter Rudolf, Johan Kronholm, for encouragement and support.