

Why Use a Unified Knowledge Representation?

John Debenham

Faculty of IT, University of Technology, Sydney, PO Box 123, NSW 2007, Australia
debenham@it.uts.edu.au

Abstract. In a unified knowledge representation, data, information and knowledge are all represented in a single formalism. A unified knowledge representation based on “items” is described. Items contain two classes of constraints that apply equally to knowledge and to data. Items are compared to an if-then, or rule-based, knowledge representation. Simple chunks of knowledge that can only be represented by a number of rules are represented as single items. Rule-based formalisms are prone to the introduction of potential maintenance hazards caused by one rule being hidden within another. A single operation for items enables some of these hidden relationships to be removed. Items make it difficult to analyse the structure of a whole application. To make the inherent structure of items clear, ‘objects’ are introduced as item building operators. The use of objects to build items enables the hidden links in the knowledge to be identified. A single operation for objects enables all of these hidden links to be removed from the conceptual model thus simplifying maintenance.

1. Introduction

A *knowledge representation* is a formalism for representing at least the data, information and knowledge things in an application. The *data* things in an application are the fundamental, indivisible things. Data things can be represented as simple constants or variables. If an association between things *cannot* be defined as a succinct, computable rule then it is an *implicit* association. Otherwise it is an *explicit* association. An *information* thing in an application is an implicit association between data things. Information things can be represented as tuples or relations. A *knowledge* thing in an application is an explicit association between information and/or data things. Knowledge can be represented either as programs in an imperative language or as rules in a declarative language. A *rule* is a chunk of knowledge in an if-then form.

A knowledge representation is *unified* if there is no formal distinction between the representation of data, information and knowledge things [1]. A unified knowledge representation for conceptual modelling is described in [2]. A *conceptual model* is a representation of an application using a particular knowledge representation. The unified knowledge representation in [2] is expressed in terms of “items” and “objects”; objects are item-building operators. The key to this unified formalism is the way in which the “meaning” of an item, called its “semantics”, is specified. A single rule of decomposition is specified for items [2]. Items are *either* represented informally as

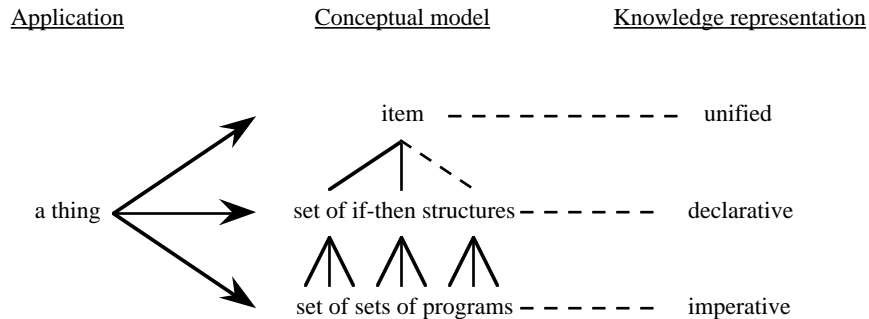


Fig. 1. A thing and its representation in the unified representation, a declarative representation and an imperative interpretation.

“i-schema” or formally as λ -calculus expressions [3]. The i-schema notation is used in applications.

A rule of inference, called item (and object) “join” is defined [4]. A single rule for “knowledge decomposition” is defined in terms of join. Knowledge decomposition may be applied to simplify the maintenance of the conceptual model and its implementation [5]. Classical database normalisation [6] is a special case of the application of that single rule. The notion of an item and the resulting conceptual model reported in [2] is extended here to fuzzy measures of knowledge integrity. This is achieved by introducing fuzzy functions to describe a graduated region of varying degrees of integrity. These fuzzy functions are generalisations of the knowledge constraints described in [3].

The expressive power of a knowledge representation must be able to describe *at least* the data, information and knowledge things. In the unified representation items also contain two classes of constraints that apply equally to knowledge and to data. In [3] these constraints are generalised to fuzzy acceptability measures of knowledge base integrity. Item and object join has been extended to apply to those measures [4].

Why use a unified knowledge representation? In the 1980s there was considerable interest in building expert systems. At that time declarative formalisms, in particular if-then formalisms such as logic programming, provided one way of computing with knowledge that was far easier to use than imperative formalisms. The comparative ease of use of if-then formalisms was responsible for the misapprehension that knowledge could be thought of as “if-then stuff”. In a sense this is true. If a chunk of knowledge has a number of if-then interpretations then it is unlikely that more than one of those interpretations will be useful at any particular time. One consequence of this misapprehension is that changes in the validity of one if-then interpretation that is not even represented may have subtle and serious implications for the validity of a number of parts of the knowledge base that are implemented. Approaches to modelling expert systems applications were based on other than declarative; for example, on frame-based systems but these are not considered here. During the ‘age of expert systems’ it was not uncommon to hear knowledge engineers observe “I took considerable trouble to build the knowledge base well but now I find that a simple change in the application can lead to an extensive maintenance task”. One reason for such an observation is that apparently useless links in the raw knowledge have been ignored. So a knowledge representation with the property that a single chunk of knowledge, which may have a number of if-then interpretations, can be represented as

a single entity could be a superior practical formalism for knowledge base design. The unified knowledge representation has this property. The unified representation is at a level of abstraction that is far closer to ‘reality’ than traditional declarative formalisms. There is a hierarchy: a real chunk of knowledge is represented as a single “item” in the unified representation. Each item has a number of interpretations as if-then forms. Each if-then form has a number of interpretations as imperative programs. This is illustrated in Fig. 1.

Further, the unified knowledge representation treats data, information and knowledge in the same way and so links between these three classes of things may also be represented. The majority of knowledge representation formalisms treat these three classes quite differently and so such links have no natural representation.

Items are compared to an if-then, or rule-based, knowledge representation. Simple chunks of knowledge that can only be represented by a number of rules are represented as single items. Rule-based formalisms are prone to the introduction of potential maintenance hazards caused by one rule being hidden within another. A single operation for items ensures that all of these hidden relationships can be removed. Items make it difficult to analyse the structure of a whole application. To make the inherent structure of items clear, ‘objects’ are introduced as item building operators. The use of objects to build items enables the hidden links in the knowledge to be identified. A single operation for objects enables these hidden links to be removed from the knowledge thus simplifying maintenance.

2. Two Difficulties with Declarative Formalisms

In a *declarative formalism* an if-then interpretation of a knowledge thing is represented in some “if-then” form. Two difficulties with declarative formalisms are illustrated using Horn clause logic as the representation formalism [7]. Logic is chosen because it is a widely understood notation, and *not* for any practical reason.

Consider the chunk of knowledge: [K1] “The sale price of a part is the cost price of that part marked up by the markup rate for that part”. This single chunk is a simple statement of fact: it is *not* in an if-then form. Under a reasonable understanding of the meaning of chunk [K1] it admits three different if-then interpretations:

$$\begin{aligned} \text{part/sale-price}(x, y) \leftarrow \text{part/cost-price}(x, z), \\ \text{part/mark-up}(x, w), y = (z \times w) \end{aligned} \quad [C1.1]$$

$$\begin{aligned} \text{part/cost-price}(x, z) \leftarrow \text{part/sale-price}(x, y), \\ \text{part/mark-up}(x, w), y = (z \times w) \end{aligned} \quad [C1.2]$$

$$\begin{aligned} \text{part/mark-up}(x, w) \leftarrow \text{part/sale-price}(x, y), \\ \text{part/cost-price}(x, z), y = (z \times w) \end{aligned} \quad [C1.3]$$

For the third of these if-then interpretations—with “part/mark-up” as its head—there is a possibility of round-off error. The three clauses [C1.1]—[C1.3] are a rather inconvenient representation of the single chunk [K1] because *one* statement of fact has been represented as *three* logical statements. Consider another chunk of knowledge: [K2] “The profit on a part is the difference between the sale price of that part and the cost price of that part.” As for [K1], the chunk [K2] is not in an if-then form. Under a reasonable understanding of its meaning, it also admits three if-then interpretations:

$$\begin{aligned} \text{part/profit}(x, y) \leftarrow \text{part/sale-price}(x, z), \\ \text{part/cost-price}(x, w), y = z - w \end{aligned} \quad [\text{C2.1}]$$

$$\begin{aligned} \text{part/sale-price}(x, z) \leftarrow \text{part/profit}(x, y), \\ \text{part/cost-price}(x, w), y = z - w \end{aligned} \quad [\text{C2.2}]$$

$$\begin{aligned} \text{part/cost-price}(x, w) \leftarrow \text{part/sale-price}(x, z), \\ \text{part/profit}(x, y), y = z - w \end{aligned} \quad [\text{C2.3}]$$

The six Horn clauses [C1.1]—[C2.3] may be combined using resolution to give some potentially useful clauses:

$$\begin{aligned} \text{part/profit}(x, y) \leftarrow \text{part/cost-price}(x, w), \\ \text{part/mark-up}(x, u), z = (w \times u), y = z - w \end{aligned} \quad [\text{C3.1}]$$

$$\begin{aligned} \text{part/profit}(x, y) \leftarrow \text{part/sale-price}(x, z), \\ \text{part/mark-up}(x, u), z = (w \times u), y = z - w \end{aligned} \quad [\text{C3.2}]$$

$$\begin{aligned} \text{part/mark-up}(x, w) \leftarrow \text{part/sale-price}(x, y), \\ \text{part/profit}(x, u), u = y - z, y = (z \times w) \end{aligned} \quad [\text{C3.3}]$$

as well as some rather useless clauses:

$$\begin{aligned} \text{part/sale-price}(x, y) \leftarrow \text{part/sale-price}(x, v), \text{part/profit}(x, u), u = v - z, \\ \text{part/mark-up}(x, w), y = (z \times w) \end{aligned} \quad [\text{C3.4}]$$

A danger with all of [C3.1]—[C3.4] is that they are assembled from particular if-then interpretations of two chunks of knowledge, namely [K1] and [K2]. If the meaning of either of those two chunks should change then [C3.1]—[C3.4] may all have to be changed as well. This is not a problem if the relationships between [C3.1]—[C3.4] and both [K1] and [K2] are represented. But in the declarative representation, it is not clear that the meanings of [K1] and [K2] are buried in the four clauses [C3.1]—[C3.4]. So [C3.1]—[C3.4] and any other clauses derived from the original six clauses above, are potential maintenance hazards [10]. Given the six original clauses [C1.1]—[C2.3] there is no reason to combine them as illustrated in [C3.1]—[C3.4]. But clauses such as [C3.1]—[C3.4] are valid and could have been constructed by a knowledge engineer as part of a knowledge base. If they form part of a knowledge-based system then they may constitute a maintenance hazard [8].

Two problems with declarative formalisms have been identified. First, they have insufficient expressive power to represent all that a single chunk of knowledge has to say. Second there is no mechanism to prevent rules from being constructed that are a potential maintenance hazard [9].

<i>part/sale-price</i>		<i>part/cost-price</i>		<i>part/mark-up</i>	
1234	1.44	1234	1.20	1234	1.2
2468	2.99	2468	2.30	2468	1.3
3579	4.14	3579	3.45	3579	1.2
1357	10.35	1357	4.50	1357	2.3
9753	12.06	9753	6.70	9753	1.8
8642	12.78	8642	8.52	8642	1.5
4321	5.67	4321	2.70	4321	2.1

Fig. 2. Value set for a knowledge item.

3. The Unified Representation: Items

Consider again the chunk of knowledge [K1]. Suppose that that chunk is represented in a formalism—which we will call an *item*—with the name $[part/sale-price, part/cost-price, part/mark-up]$. The data associated with this item may be presented as a rather messy relation; such a relation is called the *value set* of the item, a possible value set is shown in Fig. 2. An *item* consists of: an item name, item components, item semantics, item value constraints, and item set constraints. The *item name* is usually written in italics, A . The *item components* is a set of the names of items which this item represents some association between. The *item semantics* is an expression which recognises the members of the value set of this item, S_A . The *item value constraints* is an expression which must be satisfied by all members of the value set of this item, V_A . The *item set constraints* are constraints on the structure of this item’s value set, C_A . If two items have the property that the semantics of one logically implies the semantics of the other then the first item is a *sub-item* of the second. If two items have the property that they are both sub-items of each other then those two items are said to be *equivalent*.

For example, in an application there are ‘parts’ represented by the *part* data item. Each ‘part’ is identified by a ‘part-number’. The value constraint for *part* requires that part numbers should lie in the range (1 000, 9 999). The set constraint for *part* requires that there are less than 100 different part numbers. Further there are ‘costs’ represented by the *cost-price* data item. Each *part* is associated with a *cost-price*. This association is represented by the information item *part/cost-price*. That item is subject to the “value constraint” that parts whose part-number is less than 1 999 will be associated with a cost price of no more than \$300. That item is subject to the “set constraints” that every part must be in this association, and that each part is associated with a unique cost-price. The *part/cost-price* item is:

$$\begin{aligned} & part/cost-price[\lambda xy \bullet [S_{part}(x) \wedge S_{cost-price}(y) \wedge costs(x, y)] \bullet, \\ & \lambda xy \bullet [V_{part}(x) \wedge V_{cost-price}(y) \wedge ((x < 1\ 999) \rightarrow (y \leq 300))] \bullet, \\ & C_{part} \wedge C_{cost-price} \wedge (Uni(part) \wedge Can(cost-price, \{part\}))_{part/cost-price}] \end{aligned}$$

where “Uni” specifies a *universal constraint* which requires that all members of the value set of *part* must be in the value set of *part/cost-price*. “Can” specifies a *candidate constraint* which requires that members of the value set of *part* functionally determine the members of the value set of *cost-price* in this item—in traditional database jargon, part is a potential key of the part/cost-price relation. Candidate constraints for a knowledge item identify potential if-then interpretations of that item.

Items are a universal formalism in that they can describe knowledge just as naturally as information (ie relations) or data. For example, consider the chunk of knowledge [K1]. [K1] is represented as the knowledge item $[part/sale-price, part/cost-price, part/mark-up]$:

$$\begin{aligned} & [part/sale-price, part/cost-price, part/mark-up][\\ & \lambda x_1 x_2 y_1 y_2 z_1 z_2 \bullet [S_{part/sale-price}(x_1, x_2) \wedge S_{part/cost-price}(y_1, y_2) \\ & \wedge S_{part/mark-up}(z_1, z_2) \wedge ((x_1 = y_1 = z_1) \rightarrow (x_2 = z_2 \times y_2))] \bullet, \end{aligned}$$

$$\begin{aligned}
& \lambda x_1 x_2 y_1 y_2 z_1 z_2 \bullet [V_{part/sale-price}(x_1, x_2) \wedge V_{part/cost-price}(y_1, y_2) \\
& \quad \wedge V_{part/mark-up}(z_1, z_2) \wedge ((x_1 = y_1 = z_1) \rightarrow (x_2 > y_2))] \bullet, \\
& (Uni(part/sale-price) \wedge Uni(part/cost-price) \\
& \quad \wedge Can(part/sale-price, \{part/cost-price, mark-up\}) \\
& \quad \wedge Can(part/cost-price, \{part/sale-price, mark-up\}) \\
& \quad \wedge Can(mark-up, \{part/sale-price, part/cost-price\}) \\
& \quad) [part/sale-price, part/cost-price, mark-up] \\
& \quad \wedge C_{part/sale-price} \wedge C_{part/cost-price} \wedge C_{part/mark-up}]
\end{aligned}$$

Its candidate constraints identify the three if-then interpretations represented above as the clauses [C1.1]—[C1.3]. So this item represents *all* that the chunk of knowledge [K1] says. Items appear to have resolved the first difficulty with rules identified above. Items are more than a convenient way of representing a set of rules; they can be manipulated, combined and decomposed. Computing with a collection of items amounts to computing with the set of rules represented within those items so items present no greater level of computational complexity than rule-based formalisms.

4. An algebra of items

Item join is an operation that may be applied to two items A and B that share a set of common components E to construct a third item called the *join* of A and B on E , and denoted by: $A \otimes_E B$. Item join is defined formally in [2]. The following is an example of the use of the join operator:

$$\begin{aligned}
& [part/profit, part/sale-price, part/cost-price, part/mark-up] = \\
& [part/sale-price, part/cost-price, part/mark-up] \otimes \{part/sale-price, part/cost-price\} \\
& \quad [part/profit, part/sale-price, part/cost-price]
\end{aligned}$$

Using the rule of composition \otimes , knowledge items, information items and data items may be joined with one another regardless of type [11]. For example, the knowledge item:

$$\begin{aligned}
& [cost-price, tax] [\lambda xy \bullet [x = y \times 0.05] \bullet, \lambda xy \bullet [x < y] \bullet, \\
& \quad (Uni(cost-price) \wedge Can(tax, \{cost-price\}))]_{cost-price, tax}]
\end{aligned}$$

can be joined with the information item $part/cost-price$ on the set $\{cost-price\}$ to give the information item $part/cost-price/tax$. In other words:

$$\begin{aligned}
& [cost-price, tax] \otimes \{cost-price\} part/cost-price = \\
& \quad part/cost-price/tax [\lambda xyz \bullet [costs(x, y) \wedge z = y \times 0.05] \bullet, \\
& \quad \lambda xyz \bullet [((1000 < x < 1999) \rightarrow (0 < y \leq 300)) \wedge (z < y)] \bullet, \\
& \quad (Uni(part) \wedge \\
& \quad \quad Can(cost-price, \{part\}) \wedge Can(tax, \{cost-price\}))]_{part/cost-price/tax}]
\end{aligned}$$

In this way items may be joined together to form more complex items [12]. Alternatively, the \otimes operator may form the basis of a theory of decomposition in which each item may be replaced by a set of simpler items. An item I is *decomposable* into the set of items $D = \{I_1, I_2, \dots, I_n\}$ if:

- I_i has non-trivial semantics for all i ,
- $I = I_1 \otimes I_2 \otimes \dots \otimes I_n$, where
- each join is *monotonic*; that is, each term in this composition contributes at least one component to I .

If item I is decomposable then it will not necessarily have a unique decomposition. So items can be combined using the \otimes operator. The \otimes operator also forms the basis of a theory of decomposition that removes hidden relationships from the represented knowledge. So items overcome the second difficulty described above for rules.

Items are a universal formalism for knowledge representation but make it difficult to analyse the structure of the whole application. For example, two chunks of knowledge that share the same basic wisdom may be expressed in terms of quite different components; this could obscure their common wisdom [13]. To make the inherent structure of items clear ‘objects’ are introduced as item building operators [4].

5. The Unified Representation: Objects

The introduction of objects enables the structure of the conceptual model to be analysed. In [4] it is shown that objects support a formal structure which may be used to manage maintenance. Objects are item building operators which have four important properties: objects have a uniform format no matter whether they represent data, information or knowledge things; objects incorporate powerful classes of constraints; objects enable items to be built in such a way as to reveal their inherent structure, and a single rule of ‘decomposition’ can be specified for objects such that if a ‘decomposed’ set of object operators is applied to a ‘decomposed’ set of ‘base’ items then the resulting conceptual model will contain only ‘decomposed’ items and will thus be maintainable. As for items, objects may either be represented informally as “o-schema” or formally as λ -calculus expressions. Object names are written in bold italic script. For example, suppose that ***costs*** is an operator that generates the information item *part/cost-price* from its components *part* and *cost-price*; that is:

$$\textit{part/cost-price} = \mathbf{costs}(\textit{part}, \textit{cost-price})$$

Further, suppose that the object ***mark-up-chunk*** is an operator which generates the knowledge item [*part/sale-price, part/cost-price, part/mark-up*] from its components; that is:

$$[\textit{part/sale-price}, \textit{part/cost-price}, \textit{part/mark-up}] = \mathbf{mark-up-chunk}(\textit{part/sale-price}, \textit{part/cost-price}, \textit{part/mark-up})$$

In general, an n-adic object is an operator which maps n given items into another item for some value of n. Further, the specification of each object will presume that the set of items to which that object may be applied are of a specific “type”. The *type* of an m-adic item is determined both by whether it is a data item, an information item or

a knowledge item and by the value of m . The type is denoted respectively by \mathbf{D}^m , \mathbf{I}^m and \mathbf{K}^m ; unspecified, or free, type which is denoted by \mathbf{X}^m is also permitted. The definition of an object is similar to that of an item. An *object* consists of: an object name, argument types, object semantics, object value constraints, and object set constraints. The *object name* is usually written in bold italics. The *argument types* are a set of types of items to which that object operator may be applied. The *object semantics* is an expression which recognises the members of the value set of any item generated by the object. The *object value constraints* is an expression which must be satisfied by all members of the value set of any item generated by the object. The *object set constraints* are constraints on the structure of the value set of any item generated by the object. If two objects have the property that the semantics of one logically implies the semantics of the other then the first object is a *sub-object* of the second. If two objects have the property that they are each sub-objects of each other then those two objects are said to be *equivalent*.

For example, the *mark-up-chunk* object is:

$$\begin{aligned} & \mathbf{mark-up-chunk}[\lambda P:\mathbf{X}^2Q:\mathbf{X}^2R:\mathbf{X}^2\bullet\lambda x_1x_2y_1y_2z_1z_2\bullet[S_P(x_1,x_2) \wedge S_Q(y_1,y_2) \\ & \quad \wedge S_R(z_1,z_2) \wedge ((x_1 = y_1 = z_1) \Rightarrow (x_2 = z_2 \times y_2))] \bullet\bullet, \\ & \lambda P:\mathbf{X}^2Q:\mathbf{X}^2R:\mathbf{X}^2\bullet\lambda x_1x_2y_1y_2z_1z_2\bullet[V_P(x_1,x_2) \wedge V_Q(y_1,y_2) \wedge V_R(z) \wedge \\ & \quad ((x_1 = y_1 = z_1) \Rightarrow (x_2 > y_2))] \bullet\bullet, \\ & \lambda P:\mathbf{X}^2Q:\mathbf{X}^2R:\mathbf{X}^2\bullet[C_P \wedge C_Q \wedge C_R \wedge (\text{Uni}(P) \wedge \text{Uni}(Q) \wedge \text{Can}(P, \{Q, R\}) \\ & \quad \wedge \text{Can}(Q, \{P, R\}) \wedge \text{Can}(R, \{P, Q\}))]_{\vee(\mathbf{mark-up-chunk},P,Q,R)} \bullet] \end{aligned}$$

The *mark-up-chunk* knowledge object represents the essence of the knowledge within the knowledge chunk [K1] without any reference to the components of that chunk. Objects represent abstract knowledge. Objects also represent value constraints and set constraints in a uniform way. A “join” operation for objects is defined in a similar way [4] to the join operation for items described above. If a set of objects have been decomposed using the object join operator then items generated by those objects will also be in a decomposed form.

There are a number of different senses in which two objects can be considered to be “equal”. Three of these senses of “object equality” are described now. Given two n -adic objects of identical argument type $(X^{m_1}, X^{m_2}, \dots, X^{m_n})$ where each X is a type such as \mathbf{X} , \mathbf{K} , \mathbf{I} or \mathbf{D} (standing respectively for “free”, “knowledge”, “information” or “data”): $A[E_A, F_A, G_A]$ and $B[E_B, F_B, G_B]$.

A and B are *identical* written $A \equiv B$, if:

$$\begin{aligned} & (\forall Q_1, Q_2, \dots, Q_j)[E_A(Q_1, Q_2, \dots, Q_j) \Leftrightarrow E_B(Q_1, Q_2, \dots, Q_j)] \\ & (\forall Q_1, Q_2, \dots, Q_j)[F_A(Q_1, Q_2, \dots, Q_j) \Leftrightarrow F_B(Q_1, Q_2, \dots, Q_j)] \\ & (\forall Q_1, Q_2, \dots, Q_j)[G_A(Q_1, Q_2, \dots, Q_j) \Leftrightarrow G_B(Q_1, Q_2, \dots, Q_j)] \end{aligned}$$

A and B are *equal* written $A = B$, if:

$$(\forall Q_1, Q_2, \dots, Q_j)[E_A(Q_1, Q_2, \dots, Q_j) \Leftrightarrow E_B(Q_1, Q_2, \dots, Q_j)]$$

If two objects are either identical or equal then they will not necessarily have the same name.

A and B are *equivalent* written $A \simeq B$, if they are both of the same argument type and there exists a permutation π such that:

$$\begin{aligned} &(\forall Q_1, Q_2, \dots, Q_j)[E_A(Q_1, Q_2, \dots, Q_j) \Leftrightarrow E_B(\pi(Q_1, Q_2, \dots, Q_j))] \\ &(\forall Q_1, Q_2, \dots, Q_j)[F_A(Q_1, Q_2, \dots, Q_j) \Leftrightarrow F_B(\pi(Q_1, Q_2, \dots, Q_j))] \\ &(\forall Q_1, Q_2, \dots, Q_j)[G_A(Q_1, Q_2, \dots, Q_j) \Leftrightarrow G_B(\pi(Q_1, Q_2, \dots, Q_j))] \end{aligned}$$

Data objects provide a representation of sub-typing. Knowledge is quite clumsy when represented as items; objects are a more compact representation. For example, consider the *[part/sale-price, part/cost-price, part/mark-up]* knowledge item which represents the chunk of knowledge [K1]. This item can be built by applying a knowledge object *mark-up-chunk* of argument type (I^2, I^2, I^2) to the items *part/sale-price, part/cost-price* and *part/mark-up*.

6. Conclusion

A unified knowledge representation has been described. That representation is based on “items”. Items contain two classes of constraints that apply equally to knowledge and to data. Items overcome two difficulties with rule-based knowledge representations. Simple chunks of knowledge that can only be represented by a number of rules are represented as single items. Potential maintenance hazards caused by one chunk of knowledge being hidden within another that plague rule-based formalisms may be removed from the unified representation by a single operation. Items make it difficult to analyse the structure of a whole application. To make the inherent structure of items clear, ‘objects’ are introduced as item building operators. Objects are an abstraction representing the structure of knowledge. The use of objects to build items enables the hidden links in the knowledge to be identified. A single operation for objects enables these hidden links to be removed from the knowledge thus simplifying maintenance.

References

1. Barr, V. (1999). “Applying Reliability Engineering to Expert Systems” in *proceedings 12th International FLAIRS Conference*, Florida, May 1999, pp494-498.
2. Debenham, J.K. “*Knowledge Engineering*”, Springer-Verlag, 1998.
3. Debenham, J.K. “The Degradation of Knowledge Base Integrity”, in *proceedings 13th International FLAIRS Conference FLAIRS-2000*, Orlando, Florida, May 2000, pp113-117.
4. Debenham, J.K. (1999). “Knowledge Object Decomposition” in *proceedings 12th International FLAIRS Conference*, Florida, May 1999, pp203-207.
5. Debenham, J.K. “Representing Knowledge Normalisation”, in *proceedings Tenth International Conference on Software Engineering and Knowledge Engineering SEKE'98*, San Francisco, US, June 1998.

6. Date, C.J., "*An Introduction to Database Systems*" (4th edition) Addison-Wesley, 1986.
7. Mayol, E. and Teniente, E. (1999). "Addressing Efficiency Issues During the Process of Integrity Maintenance" in *proceedings Tenth International Conference DEXA99*, Florence, September 1999, pp270-281.
8. Ramirez, J. and de Antonio, A. (2000). "Semantic Verification of Rule-Based Systems with Arithmetic Constraints" in *proceedings 11th International Conference DEXA2000*, London, September 2000, pp437-446.
9. Katsuno, H. and Mendelzon, A.O., "On the Difference between Updating a Knowledge Base and Revising It", in *proceedings Second International Conference on Principles of Knowledge Representation and Reasoning, KR'91*, Morgan Kaufmann, 1991.
10. Johnson, G. and Santos, E. (2000). "Generalizing Knowledge Representation Rules for Acquiring and Validating Uncertain Knowledge" in *proceedings 13th International FLAIRS Conference*, Florida, May 2000, pp186-2191.
11. Werner, E. "Logical Foundations of Distributed Artificial Intelligence" in O'Hare, G.M.P. & Jennings, N.R. (Eds) "*Foundations of Distributed Artificial Intelligence*", pp57-118, Wiley, 1996.
12. Darwiche, A. (1999). "Compiling Knowledge into Decomposable Negation Normal Form" in *proceedings International Joint Conference on Artificial Intelligence, IJCAI'99*, Stockholm, Sweden, August 1999, pp 284-289.
13. Jantke, K.P. and Herrmann, J. (1999). "Lattices of Knowledge in Intelligent Systems Validation" in *proceedings 12th International FLAIRS Conference*, Florida, May 1999, pp499-505.