Weakly constraining multimedia types based on a type embedding ordering

Christian Altenschmidt, Joachim Biskup, Jürgen Freitag, Barbara Sprick

Fachbereich Informatik, Universität Dortmund, Germany
{altensch|biskup|jf|sprick}@ls6.informatik.uni-dortmund.de

Abstract. We present a concept of weakly constraining types which balances heterogeneity and fixity of data structures. This concept is designed for a multimedia mediator that uses fixed type declarations on schema level but allows variations of actual structures on instance level. The concept is based on a notion of embedding a fixed type declaration into a variation structure such that essential aspects of the fixed declaration are preserved. Finally we show how multimedia types gain from our type system.

1 Introduction

1.1 Balancing heterogeneity and fixity of structure

In the evolving information society users increasingly expect informational services that can deal with various kinds of heterogeneity. At the same time traditional database schemas fix the structure of stored data by declaring its types. When designing advanced information systems we are faced with the challenge of integrating the benefits of heterogeneity and the advantages of fixing the structure of data. In this paper we report on the fundamentals of a specific solution to this challenge. This solution is part of our overall design of a *multimedia mediator* (MMM), that is currently implemented in a prototype fashion [5, 6].

From the perspective of a user, the MMM provides a well-structured gateway to some application dependent part of a federated multimedia system. Seen from inside, the MMM employs proxy objects for external multimedia items, hides all details of external communication, and its query optimization takes advantage, among other things, of materializing previous query answers in the proxy state. We intend the MMM to be set up in a top-down approach. Based on the analysis of an application, an administrator first declares a schema tailored for potential users. Then conditions for the access to external sources are specified, partly automatically and partly on a case by case basis. Later on, whenever the users refer to proxy objects, the system transparently supplies all translations that are necessary to access the actual sources.

1.2 Design of a type system

The services of the MMM are based, among others, on a new concept of weakly constraining types that constitute our specific solution to allow restricted heterogeneity while partially fixing the structure of data. Our new concept is part of the intended type system of the MMM. The mediator should automatically supply certain standard translations between fixed type expressions in the schema and various actual structures on the instance level. More special kinds or specific aspects of translations have to be individually supplied by appropriate wrappers on a case by case basis.

In this paper we define a suitable notion of *embedding* one type into another. Strictly speaking, we should always clearly distingish the syntactic and semantic aspects of a type, i.e. the (syntactic) type expression, as occuring in a schema, and its (semantic) extension which is defined as the set of complying instances. Furthermore, even if a particular data item is not statically considered to be a member of the type extension, it nevertheless can explicitly or implicitly have a structure which might comply with the type expression. In that case we can reasonably speak about the data item having that type as a dynamic type. Though being important for our considerations [8], in the following these distinctions are often blurred by abuse of language, as usual in computer science.

Our notion of an embedding is expressed by a function that maps nodes of (the syntax tree of) a first type (expression) onto nodes of (the syntax tree of) a second type (expression). Roughly speaking, this function indicates how each edge of the first type is translated into a path of the second type. Based on this definition of type embedding, we introduce a new type constructor **self_describe** that can be used for partial structural descriptions on schema level. This constructor associates a type on schema level with a set of structures on instance level such that the schema level type expression can be embedded in each of the instance level structures.

1.3 Achievements and comparison

Recently a lot of research on semi-structured data has been done (cf. [1, 8, 12]). Many approaches adopt the Object Exchange Model (OEM) to describe heterogeneous objects, and corresponding schemas [9, 11] essentially describe all possible paths to (components of) objects.

Two recent implementations of dbs with semi-structured objects, Lore [11] and the OQL-doc project [2], can be seen as extensions of the ODMG proposal. Lore meets the problem of object heterogeneity by avoiding typing, while OQL-doc explicitly enriches the type system with a union type. To overcome the differences in data structure resulting from heterogeneity many approaches, like [10, 2, 3], employ regular path expressions. Lorel [3] uses coercion for comparing objects of different types. In UnQL coercion is avoided by exploiting pattern matching [10].

The main difference between the approaches mentioned above and our concept is our top down approach to mediation. We design a uniform schema, describing all essentials by a type definition that ignores heterogeneity. Queries are well-typed and related to this uniform view, thus they can be answered transparently without the need of regular path expressions. We define when values can be translated into our uniform view. We argue, that this is possible in an algorithmically generic way, if we can apply some kind of natural coercion [7], namely a mapping of "finer" values onto "coarser" values. This way a type union is not explicitly needed and we are open to add new structures without the need of changing the schema.

2 A new type system and its requirements

We are looking for a specific approach to maximize the allowed heterogeneity of data while sufficiently fixing its structure. We propose to introduce a new kind of types that allow variations in the structure of data instances (for the sake of heterogeneity), but also insist to specify *essential constraints* on them (for the sake of semantic clarity and computational efficiency).

2.1 A new type system

We design a new type system with the following goals in mind: The type system should potentially *integrate a broad spectrum* of types. It should *deal with semi-structured data* and it should support the intended *top-down usage* of the MMM. For meeting well-typedness, a *path expression* should correctly and efficiently be interpreted on the instance level which may be spread across various sources.

With these overall goals in mind, our concept of *weakly constraining types* is devised to supply certain standard translations for reasonably manageable situations. On schema level, we will interprete certain *types* as time-independent partial structural descriptions of the anticipated data instances. On instance level, any single data item has a particular, possibly time-dependent structure that must *conform* to the schema type. Then, for any data item the raw data is supplemented with some kind of self-description consisting of its particular structure and the relationship between this actual structure and the schema type.

2.2 Objects as abstractions of reality

Our MMM is based on maintaining proxy objects for external multimedia items. A first set of requirements is derived from the following interpretation of values of proxy objects as *abstractions* of the external multimedia items.

First we emphasize that a multimedia item and its representing proxy object refer to the same "entity of the real world". A multimedia item is a database object identified by a unique object identifier and containing a description of the entity as its value. Thus the multimedia item is designed as an abstraction with respect to the essential aspects of the world. The corresponding proxy object further abstracts from the multimedia item.

How can we produce abstractions? Basically, we abstract by *removing actual information* from the original object, for instance by neglecting certain aspects, making data coarser or aggregating data. We argue that to some extent removing information can be accomplished in an algorithmically generic way. However, if we want to *add conceptual information* we cannot do so in an algorithmically generic way but have to employ case specific wrappers.

We argue that we could remove arbitrary information, but intuitively the removal should preserve a certain property of object relationship, namely, whenever the values of two objects are indistinguishable before the removal they should still be indistinguishable afterwards.

2.3 Essential constraints

As mentioned above the mediator administrator specifies essential constraints to be placed on the source objects by declaring a type on schema level, i.e. he defines a *tree* with inner nodes describing the desired hierarchical structure and leaves describing the desired content data.

Now we look for a mechanism to actually place these essential constraints on source objects. We know that a source object is primarily restricted by its particular (dynamic) type, no matter whether this type is defined explicitly in the source level schema or generated adhoc by a wrapper. So it suffices to place the essential constraints on this type. We do this by demanding that an *embedding function* i from the nodes of the schema level type tree t_1 into the nodes of the instance level type tree t_2 exists. This function should have two additional properties described in the following.

We want to guarantee that all atomic components of the first type (identified with) t_1 are also present as *independent* components in the second type (identified with) t_2. Then, the flat information contained in a data unit of type t_1 can also be expressed by a data unit of type t_2. More formally, function i *discriminates leaves*, i.e. i is *injective* on the set of leaves of t_1 in such a way that for any two leaves n and m of t_1 the nodes $i(n)$ and $i(m)$ are *not connected* in t_2. Note that by this definition it is possible to map leaves onto inner nodes, which allows atomic values on schema level to be structurally more differentiated on instance level.

We want to ensure that the internal structure of the first type t_1 is also somehow represented in the second type t_2. Then, the structural information contained in a data unit of type t_1 can also be somehow expressed by a data unit of type t_2. More formally, function i *strictly preserves connections*, i.e. for each edge (n, m) in t_1 there exists a corresponding *nonempty* path in t_2 connecting the nodes $i(n)$ and $i(m)$. As an immediate consequence, also for each path in t_1 from node n to node m there exists a corresponding path in t_2 connecting the nodes $i(n)$ and $i(m)$.

2.4 Requirements for path expressions

Having discussed the type system from a conceptual point of view we will now specify some more pragmatic requirements. For query evaluation we would like to query source objects as if they were of a type defined in the schema. This means it is necessary to correctly and efficiently interprete path expressions being well-defined with respect to the essential constraints specified in the schema. More precisely, a path in a constraint type should be *faithfully* translated into a path in a variation type, i.e. if a path p_1 in the constraint type is a strict initial part of a path p_2 then the translation of p_1 is a strict initial part of the translation of p_2. Summarizing, the embedding should maintain the structural data of a constraint type also in the variation types, and it should allow conversion of content data in a variation instance into content data for the constraint instance.

3 An embedding ordering and its algebraic properties

We build our type system from some atomic types, e.g. **boolean**, **integer** and **string**, by applying some type constructors, e.g. **n_tuple_of**, for cardinals n, **set_of**, and **reference_to**. For all scalar values, we will omit an explicit constructor **scalar**. Any type σ is described by its syntax tree, where the leaf nodes denote the involved atomic types and the nonleaf nodes denote the use of type constructors. Thus a type σ can be considered as a pair $\langle temp, label \rangle$, where *temp* describes the pure graph theoretical structure of the syntax tree, and *label* describes the annotations at the nodes of the syntax tree. Figure 3 shows a simple example of a type in different

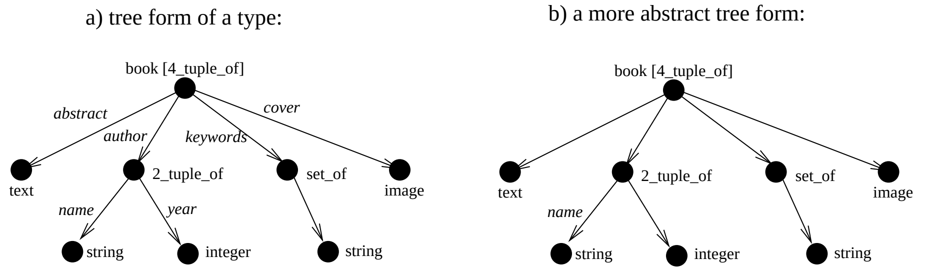

Fig. 1. Different kinds of type presentation

forms. Part a) depicts a type in tree form. Attribute names appear as labels on the edges. Part b) is derived from part a) by omitting the attribute names. Later on, for the convenience of presentation, we will mostly use this kind of presentation.

In this context, we assume that for any nonleaf node of any tree the descending edges are marked with (locally unique) identifiers. Thus for any node with k successors we can distinguish the corresponding k subtrees by these marks even if they are isomorphic. We will, however, omit the identifiers for the sake of succintness.

If we understand values as abstractions, a value of type σ is seen as an equivalence class of values of type τ, namely of those τ-abstractions that are coarsened to the σ-abstraction. For atomic types this means in terms of a type system that an atomic type σ is contained in τ, $\sigma \leq \tau$, iff the domain of σ is considered as a subdomain of τ. For example, we specify that **boolean** $\leq$ **integer** $\leq$ **string**. If τ is a complex (= structured) value, we still demand at least $|\sigma| \leq |\tau|$.

Abstractions of constructors are coarsened by deleting components. In terms of a type system this means, a constructor c_2 is a refinement of a constructor c_1, $c_1 \leq c_2$, iff all components of c_1 are also components of c_2. For instance, considering tuple constructors, we can specify that for $n \leq m$, **n_tuple_of** $\leq$ **m_tuple_of**. Note, that the constructor **m_tuple_of** can also be simulated by a cascade of tuple constructors each of which may have an arity less than m.

Essential constraints as mentioned in section 2.3 are placed on the type trees as a whole. Their effects on our embedding will be more formally investigated in the next subsections.

3.1 Types involving only one atomic type and tuple constructors

We first treat the special case that we have only one atomic type *atom* and only the tuple constructors **n_tuple_of**, for all cardinals n.

Definition 3.1 (embedding of (type) trees) *A tree $temp_1$ is* embeddable *into a tree $temp_2$, $temp_1 \leq temp_2$, iff there is a mapping i of the nodes of tree $temp_1$ on the nodes of tree $temp_2$ such that the following two properties hold:*

1) Mapping i discriminates leaves *(see section 2.3).*

2) Mapping i strictly preserves connections *(see section 2.3).*

Lemma 3.1 *The embedding $\leq$ is* closed for subtrees*: Suppose $temp_1 \leq temp_2$ via some mapping i, and consider the subtree $temp_1,n$ determined by a node n of type $temp_1$ and the subtree $temp_{1,i(n)}$ determined by the corresponding node $i(n)$. Then also $temp_{1,n} \leq temp_{1,i(n)}$ via the same mapping i.*

Lemma 3.2 *The embedding $\leq$ is reflexive and transitive.*

To proof transitivity, we argue that if an embedding maps a leaf of $temp_1$ onto an inner node of $temp_2$ we can find an equivalent embedding which maps the leaf of $temp_1$ onto a leaf of $temp_2$, such that we gain injectivity on leaves.

So our definition of embedding $\leq$ turns out to result in the useful algebraic structure of a partial ordering. Here we define two trees to be in the same equivalence class iff they are identical except of a systematic (bijective) renaming of their nodes and edges. We always take a tree as a representative member of its equivalence class. With this convention in mind we can state the following theorem.

Theorem 3.1 *The embedding $\leq$ is antisymmetric, and thus the embedding $\leq$ is a partial ordering on types.*

Proof (sketched) We first show that we can choose the mappings i and j as inverse functions on the leaves. Then it is a fact that a rooted tree is completely characterized by the set of its leaves and the ancestor function. With this we can proof (by induction on the distance from leaves) that the ancestor function of tree $temp_1$ is equal (up to renaming) to the ancestor function of tree $temp_2$. □

In section 2.2 we have treated values as abstractions and discussed their property of being indistinguishable. Let us now define that two abstractions are *indistinguishable* if their trees can be mutually embedded such that contents of corresponding leaves are equal. Then theorem 3.1 can also be interpreted as follows.

Theorem 3.2 *The embedding $\leq$ preserves indistinguishability of abstractions as stated in section 2.2.*

3.2 Impact of additional atomic types and type constructors

We now relax the assumed restrictions, namely that we have only one atomic type *atom* and only the tuple constructors **n_tuple_of**, for all cardinals n. The first restriction can be dropped as follows. Remember that, by the construction principle of types, atomic types only appear as leaves and that, by the property of discriminating leaves, leaves are injectively mapped on independent subtrees. We first discuss the special case where these subtrees are leaves again. If we now assume that we have several different atomic types, then, in all cases, we only have to take care that any leaf type t_1 is mapped on a leaf type t_2 such that, considered as atomic types, $t_1 \leq t_2$, i.e. t_1 is contained in t_2. When we deal with "semantic domains", i.e. syntactic types with specialized operations, or the general case that a leaf is mapped on a nontrivial subtree, the notion of containment might be quite subtle and is subject to semantically motivated decisions to be taken by the mediator administrator.

Unfortunately, when differentiating atomic types our embedding loses transitivity. This results from the possibility of mapping leaves onto inner nodes combined with our demand for containment of the atomic type of the leaf in the complex value of the inner node. For instance, in 2 we can neither map both v_1 and v_2 on w since this mapping would not discriminate leaves, nor can we guarantee that both v_1 and v_2 are contained in one of the leaves in σ_3. For now we have to leave this problem open.

For relaxing the second restriction we have to deal with the specific properties of any additional constructor. For the sake of space we just mention that we can appropriately treat two powerful constructors that appear in object-oriented databases, namely the set constructor **set_of** and the reference constructor **reference_to**. Further constructors still need to be investigated, see also section 5.

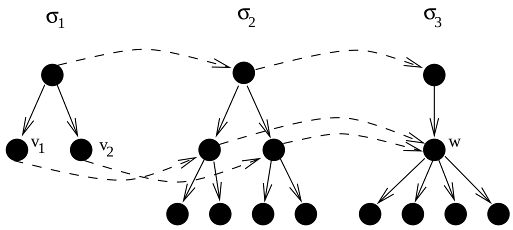

Fig. 2. An example for loss of transitivity

Summarizing, we can formally define a generalized notion of type embedding. For an example, see [6]. This notion is based on two more fundamental definitions:
- any given definition of containment $\leq$ of atomic types
- the specific definition of refinement $\leq$ of the type constructors **n_tuple_of**, **set_of**, and **reference_to** that comprises only the trivial containments **set_of** $\leq$ **set_of** and **reference_to** $\leq$ **reference_to** as well as, for $n \leq m$, **n_tuple_of** $\leq$ **m_tuple_of** as already stated in the introduction of this section.

Definition 3.2 (embedding ordering on types) $\langle temp_1, label_1\rangle \leq \langle temp_2, label_2\rangle$*: iff* $temp_1 \leq temp_2$ *via a mapping* i *(satisfying the conditions given in definition 3.1), and* $label_1(n) \leq label_2(i(n))$*, for all nodes* n *of* $temp_1$ *such that* $label_1(n)$ *is not a tuple constructor.*

4 A concept of weakly constraining types

We are now ready to introduce our new concept of weakly constraining types. For this goal we propose a new type constructor **self_describe** with two parameters. The first parameter must be a usual type specifying the time-independent essential constraints on the structure of allowed instances. Any processing of data has to be well-typed with respect to this type, possibly by using an embedding.

The second parameter, a *meta-type* the values of which are types, is used to restrict the allowed instances more concretely. Both parameters must be *compatible* in the sense that the first parameter can be embedded in all types given by the second parameter. Some data unit is allowed as an instance iff the particular structure of the data unit (i.e. its dynamic type) is an element of the second parameter.

For any type σ occuring as a first parameter, we can use the corresponding meta-type Σ, consisting of all types τ such that σ can be embedded into τ, as a default for the second parameter. Again, the reader is referred to [5, 6] for an example.

5 Embedding multimedia types

Starting with traditional types as described in section 3 we can sketch some multimedia types and discuss reasonable embeddings by some simple examples. A first characteristic type for multimedia systems is *text*, which differs from plane strings by including structural information. Following our argumentation for traditional types we would like to embed a less structured text in a more structured one, and viewing a string as a text without any structure we define $string \leq text$.

Images stored in a database are usually enriched by visual features to allow a content-based retrieval. Thus we may view an image as a tuple consisting of the image itself and additional attributes covering information like colour and texture, or even a set of identifiable objects together with their spatial interrelationships. We again can take over definition 3.2 leading to common-sense emdeddings of different image types, where the coarser image type lacks some attributes and therefore some kind of information. Seeing colour as meaningful information, we may define $greyscale\ image \leq colour\ image$. Finally, in the presence of a process which detects text within an image, we can argue $string \leq image$.

The most significant features of multimedia data result from its representation and demand the incorporation of time-related concepts. The main ingredients are composition principles like sequential composition and parallel composition. When talking about embedding we would expect information of time to be removed in the coarser type, which is introduced by two simple examples: a sequential constructor may arrange images to be shown one after one with a certain delay, and the set of images is embedded in the sequential arrangement. Likewise a parallel constructor synchronizes the presentation of different objects to be shown in parallel, and the tuple of objects is embedded in its parallel arrangement.

References

1. S. Abiteboul. Querying semi-structured data. In Afrati and Kolaitis [4], pages 1–18.
2. S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Simon. Querying documents in object databases. *Journal of Digital Libraries*, 1(1):5–19, 1997.
3. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1):68–88, 1997.
4. F. Afrati and P. Kolaitis, editors. *6th Int. Conf. on Database Theory*, Delphi, Greece, Jan. 1997. Springer-Verlag.
5. J. Biskup, J. Freitag, Y. Karabulut, and B. Sprick. A mediator for multimedia systems. In *Proc. 3rd Int. Workshop on Multimedia Information Systems*, pages 145–153, Como, Italy, Sept. 1997.
6. J. Biskup, J. Freitag, Y. Karabulut, and B. Sprick. Query evaluation in an object-oriented multimedia mediator. In *Proc. 4th Int. Conf. on Object-Oriented Information Systems*, pages 31–43, Brisbane, Australia, Nov. 1997.
7. K. B. Bruce and G. Longo. *A modest model of records, inheritance, and bounded quantification*, pages 151 –195. MIT Press, Cambridge, London, 1994.
8. P. Buneman. Semistructured data. In *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 117–121, Tucson, AZ, May 1997.
9. P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In Afrati and Kolaitis [4], pages 336–350.
10. G. Fernandez, L. Popa, and D. Suciu. A structure-based approach to querying semi-structured data. In *Proc. Workshop on Database Programming Languages*, 1997.
11. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.
12. D. Suciu. Management of semistructured data. *SIGMOD Record*, 26(4):4–7, 1997.