Term Subsumption with Type Constructors

Barbara Piza, Klaus–Dieter Schewe, Joachim W. Schmidt

University of Hamburg, Dept. of Computer Science,
Vogt-Kölln-Str. 30, D-W-2000 Hamburg 54
{piza|schewe|J_Schmidt}@dbis1.informatik.uni-hamburg.de

Abstract

Term Subsumption Languages (TSLs), a generalization of both semantic networks and frames equipped with a model-theoretic semantics, have shown to be of vital practicability for the representation of knowledge. Recently it has been claimed that the modelling power of TSLs could be decisively enhanced by introducing type constructors as they are used in modern typed programming languages.

In this paper we do a first step into this direction regarding only those type-constructors that are commonly used in semantic data models (SDMs). Due to the similarity of abstraction mechanisms in SDMs and TSLs the development of the integrated model called IFO$^+$ is straightforward. Moreover, it turns out that inference mechanisms known in both disciplines can also be unified on the basis of IFO$^+$. In particular subsumption can be regarded as a stronger version of absolute dominance.

keywords: term subsumption language, semantic data modelling, subsumption, type constructor, restructuring

1 Introduction

The notion of *term subsumption languages* (TSLs) was introduced by Peter Patel-Schneider in order to refer to all kinds of knowledge representation systems based on Brachman's original idea of KL-ONE [7]. Systems such as KRYPTON [6], BACK [10, 11] and CLASSIC [4] have emerged from this idea. These systems generalize both semantic networks and frames. Moreover, they embody a formal *term description language* together with a model-theoretic semantics.

In general TSLs distinguish between a *terminological* layer consisting of a generalization/specialization hierarchy of types and an *assertional* layer of individual members of such types. Moreover, types in such a hierarchy can be related via attribute functions. In TSLs types are usually called *concepts* and attributes are called *roles*. Concepts and roles can be defined by terms. The important additional feature of TSLs that makes them differ fundamentally from semantic networks or frames is the possibility to introduce *definitional terms*, hence the need to compare terms by a special inference mechanism called *classifier*.

The task of the classifier is to take term definitions, to compare them and to decide whether one is more general than the other. A second task would be to derive the concept of a given definition of an individual object. The latter task is sometimes treated seperately by an inference mechanism called *recognizer*.

Type generalization/specialization hierarchies are also used in *semantic data models* (SDMs) [9]. A prominent representative of SDMs is the IFO model [1] due to its genericity in the sense that almost all known SDMs are representable in IFO. A schema in IFO consists of a hierarchy of classes, where specialization by introducing subclasses differs from generalization by union construction, and attribute functions relating classes. These attributes are called *fragments* in IFO. Moreover, classes can be built using an aggregation- and a group-constructor. The important feature of IFO that raises it beyond the functionality of other SDMs is that we can associate with each class in a schema a derived type thereby introducing disjoint union as a third type constructor. Derived types form the basis of an inference mechanism called *restructuring*.

The role of the restructuring inference mechanism is to derive a normal form representation such that types have the same information capacity iff they have the same normal form. Fragments and specialization relations are not used within restructuring.

Both models share the same abstraction mechanisms, i.e. classification, generalization and functional attributes. The difference between the two models is the distinction between primitive and defined concepts in TSLs and the use of type constructors in IFO. Moreover, both inference mechanisms are based on term rewriting techniques. The difference is that classification in TSLs exploits definitional concepts and restrictions on roles, whereas restructuring uses algebraic relations between nested type constructors. Hence the idea to unify both models.

From an IFO perspective this means to include fragments into derived types and to introduce a distinction between primitive – those associated with specializing classes – and defined types – those we had before. From a TSL perspective this means to introduce three type constructors (disjoint union, aggregation, grouping).

Then it should be possible to define a set of rewriting rules that extends both the restructuring rules and the rules used by the classifier. Consequently the notions of absolute dominance, i.e. of extended information capacity, and of subsumption become comparable in a uniform framework. One of our results states that subsumption is more special than absolute dominance, since it assumes the preservation of the structure.

Recently it has been claimed that it should be possible to enhance TSLs even by more complex constructors such as a polymorphic function constructor [3]. In this paper we only treat the unification of TSLs with IFO. In section 2 we introduce the unified data model which we call IFO$^+$. Moreover, we formally define the notions of subsumption and absolute dominance. In section 3 we describe our set of restructuring rules on the basis of IFO$^+$. This set comprises normalization rules that stem from TSLs, capacity preservation rules from IFO and reduction rules for a smooth integration of both. In section 4 we outline the Church-Rosser property of our set of restructuring rules. This enables us to characterize absolute dominance and subsumption in terms of normal forms. We conclude with a short summary and outlook in section 5.

2 The IFO$^+$ Model

In almost all TSLs we assume to be given a predefined concept ANY. Then a schema can be built successively by adding concepts that restrict previously introduced concepts, where each concept is introduced by giving its supertypes and restrictions to the roles. Moreover, we may distinguish between *primitive* and *defined* concepts depending on whether role restrictions

are regarded as necessary conditions or as necessary and sufficient conditions respectively. Throughout this paper we will assume a rather simple TSL without multi-valued roles and cardinality constraints.

In SDMs such as IFO we use three type constructors, i.e. disjoint union, (finite) sets and tuples. Complex types are constructed from simple types using these constructors. Clearly the nesting of constructors is allowed. Then the general problem in unifying TSLs with IFO is to add these constructors to our basic TSL. This will give the IFO$^+$ model which will be described in the following. A more detailed and formal description of IFO$^+$ can be found in [12].

2.1 Syntax of IFO$^+$-Schemata

As described above a TSL schema should be built successively starting with a predefined type ANY. The same applies to an IFO$^+$ schema. Each new type in a TSL is built using supertypes and role definitions. Roles can be new or inherited from one of the supertypes. The only extension to type definitions in IFO$^+$ is the introduction of an optional structural component, where each structure can be built using (nested) tuple-, set- and union-constructors.

In TSLs we distinguish between primitive and defined types. In principle the same applies to IFO$^+$. However, types with a structure part should not be allowed to be primitive. Hence the following syntax of IFO$^+$. We use the symbol $\uparrow$ to denote supertypes. We use $[\cdot]$, $\{\cdot\}$ and $< \cdot >$ to denote the tuple-, (finite) set- and (disjoint) union-constructors respectively. Note that the tuple- and the union-constructor may have an arbitrary arity $n > 0$ whereas the set-constructor has arity 1. Moreover, we use the usual meta-symbols { } and [ ] to denote repeated and optional occurrences. | is the meta-symbol used for selection. We assume to be given a countably infinite set of type-names and a countably infnite set of role-names.

schema-declaration	::	{ type-declaration }$^+$
type-declaration	::	primitive-type-declaration | defined-type-declaration
primitive-type-declaration	::	type-name ":<" "↑"({ type }) ({ role-declaration })
defined-type-declaration	::	type-name ":=" "↑"({ type }) [structure-declaration] ({ role-declaration })
type	::	ANY | type-name | type-declaration
structure-declaration	::	"[" { type }$^+$ "]" | "{" type "}" | "<" { type }$^+$ ">"
role-declaration	::	role-name ":" type

We assume that types occuring in a type-declaration have been introduced earlier in the sequence defining an IFO$^+$ schema, hence there is an ordering on type-declarations within a schema-declaration. If S is a schema, let $\mathcal{F}(S)$ denote the set of role-names occurring in S.

2.2 Relation to IFO and TSLs

An IFO$^+$ schema S such that for all defined types in S the structure-declaration part is empty is in fact a (simple) TSL schema. Hence the basic features of TSLs [5, 7, 10] are representable in IFO$^+$. Moreover, the derived types of an IFO schema [1] are also directly representable

in IFO$^+$. This can be achieved if we omit all role-declarations. In addition, the only allowed supertype is ANY and this may only occur within the declaration of primitive types.

A collection of types in a TSL uniquely defines the schema. In the case of IFO the schema is seperated from the derived types. However, it is possible to represent also an IFO schema as an IFO$^+$ schema. In an IFO schema we have two kinds of basic types, called *printable* or *abstract*. Printable types correspond to basic data types such as $Integer$, $String$ etc. Abstract types correspond to collections of real world entities such as $Person$, $Boat$ etc. The only difference is that we may associate attributes (fragments, roles) with abstract types but not with printable types. Hence both kinds of types can be represented by primitive types in IFO$^+$ without supertypes other than ANY and with a role-declaration part that is either empty in the case of printable types or non-empty in the case of abstract types.

Composed types in an IFO schema are built from the three type constructors. Their representation in an IFO$^+$ schema is simply given by a corresponding defined type declaration without any supertype other than ANY.

Free types in an IFO schema are twofold. They may occur either as generalizations or as specializations of existing types. Hence they have to be represented by a defined type with a disjoint union structure or by a primitive subtype respectively.

Type derivation in IFO replaced free types occurring as generalizations by union types. This becomes obsolete in IFO$^+$ because the disjoint union constructor is part of a schema definition. In IFO the derived type of a free type occurring as a specialization is the same as the derived type of the supertype. Schema design rules prevent the diamond problem to be relevant. The approach in IFO$^+$ is a little bit more sophisticated, since we now associate with such a free specialization a primitive subtype. Moreover, type derivation in IFO omits all fragments whereas in IFO$^+$ roles are kept within types.

2.3 A Sample IFO$^+$-Schema

Here the restructuring example from [1] is shown, the syntax is illustrated in figure 1, the subsumption vertices to the ANY node are omitted for simplicity.

Name	:< ↑ (ANY)
Hull	:< ↑ (ANY) (license: ANY)
Motor	:< ↑ (ANY)
Car	:< ↑ (ANY) (motor: Motor)
Motor-Boat	:= [Motor Hull]
Vehicle	:= <Motor-Boat Car> (capacity: ANY)
Person	:< ↑ (ANY) (p-name: Name)
Owner	:< ↑ (Person) (owns: Vehicles := {Vehicle})
Boat-Owner	:= ↑ (Person) (owns: Boats := {Motor-Boat})

2.4 Semantics of IFO$^+$-Schemata

So far we have only defined the syntax of an IFO$^+$ schema. We have seen that it smoothly extends both IFO and TSLs. We want to extend our approach giving a model-theoretic semantics for IFO$^+$ extending the one for TSLs [5, 7, 10] and for IFO [1, 2, 9]. The general TSL approach starts with an arbitrary countably infinite set D. Each type gets mapped to a subset of D and each role gets mapped to a subset of $D \times D$. The mapping is usually denoted

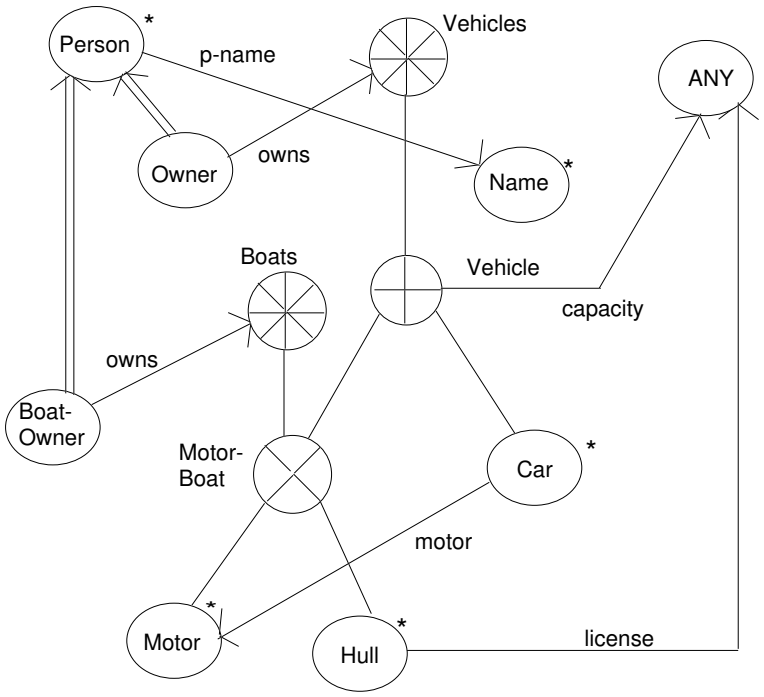

Figure 1: Sample Schema

ϵ in both cases. Hence a type corresponds to an immutable set of possible values. Moreover, we have $\epsilon(A) \subseteq \epsilon(B)$ if A is a subtype of B. On the level of a schema we associate with each type a finite subset of the set of possible values. This set is mutable.

In SDMs, especially in IFO, we associate arbitrary countably infinite and pairwise disjoint sets with our given base types. Composed types get mapped to cartesian products, disjoint unions or (finite) power sets respectively. Again we receive immutable sets of possible values. On the schema level we associate mutable subsets with types and mutable sets of pairs with fragments.

Hence the introduction of type-constructors raises the following problem. The semantics of types in a TSL schema is usually given in a restrictive way starting from one set and building successively subsets whereas IFO uses a constructive approach starting from a collection of base sets and applying general constructors. For a smooth integration of both approaches we have to use a simple trick. We start with one arbitrary countably infinite set D_0 and first construct a domain D over D_0 that contains D_0 and is closed under cartesian products, disjoint unions and finite subset building. We omit the formal details [12]. Then we define $\epsilon(ANY) = D$ and define the semantics $\epsilon(T)$ of each type T in an IFO$^+$-schema S as a subset of D. Moreover, if r is a role in S, then $\epsilon(r)$ is a subset of $D \times D$ just as in the case of TSLs.

More precisely, if S is an IFO$^+$-schema and T, T' are type-declarations in S, then we write $T \sqsubset T'$ iff T occurs before T' in S. Hence for each role $r \in \mathcal{F}(S)$ there exists a minimal T such that $r : T'$ occurs in the role-declaration part of T and $T' \sqsubset T$. Then we can define $\epsilon(r) = FUN(\epsilon(T), \epsilon(T'))$. Here $FUN(A, B)$ denotes the set of functions from A to B. Then ϵ is defined on types as follows:

- If $T :< \uparrow (T_1 \ldots T_n)(r_1 : T'_1 \ldots r_m : T'_m)$ is a primitive type declaration, then we define

$$\epsilon(T) \subseteq \epsilon(T_1) \cap \ldots \cap \epsilon(T_n) \cap \epsilon_T(r_1 : T'_1) \cap \ldots \cap \epsilon_T(r_m : T'_m)\text{ , where}$$

$$\epsilon_T(r_i : T'_i) = \begin{cases} \{d \in D \mid \exists d' \in \epsilon(T'_i).(d, d') \in \epsilon(r_i)\} & \text{if } r_i \text{ occurs in some } T' \sqsubset T \\ D & \text{else} \end{cases}$$

- If $T :=\uparrow (T_1 \ldots T_n)\, struct\, (r_1 : T'_1 \ldots r_m : T'_m)$ is a defined type declaration, we define

$$\epsilon(T) = \epsilon(T_1) \cap \ldots \cap \epsilon(T_n) \cap \epsilon(struct) \cap \epsilon_T(r_1 : T'_1) \cap \ldots \cap \epsilon_T(r_m : T'_m)\text{ , where}$$

$$\begin{array}{lll} \epsilon([T_1 \ldots T_n]) & = & \epsilon(T_1) \times \ldots \times \epsilon(T_n) \quad \text{(cartesian product)} \\ \epsilon(< T_1 \ldots T_n >) & = & \epsilon(T_1) \dot{\cup} \ldots \dot{\cup} \epsilon(T_n) \quad \text{(disjoint union)} \\ \epsilon(\{T'\}) & = & \wp_0(\epsilon(T')) \quad \text{(finite subsets)} \end{array}$$

  and ϵ_T is defined as in the primitive case.

This model-theoretic semantics gives us only the set of possible values for each type T in a schema S. In order to complete the definition of semantics for IFO$^+$ we have to define the notion of an *instance*. Basically an *instance* of S is a mapping *inst* with $inst(T) \subseteq \epsilon(T)$ for each type T and $inst(f) \subseteq \epsilon(r)$ for each role r. All these subsets are finite and satisfy the constraints of the schema. We omit the formal details [12].

3 Restructuring in IFO$^+$

We have shown that the IFO$^+$ model structurally extends both a basic TSL and IFO, but this is only part of the whole story. In order to become a vital extended term subsumption language we have to show that classification [10] is preserved within IFO$^+$. Moreover, there is also an inference mechanism based on IFO which is called restructuring [1, 2]. Therefore, we have to define an inference mechanism on top of IFO$^+$ that simultaneously extends restructuring and classification.

Both theses inference mechanisms can be based on term rewriting plus a follow-on analysis of the resulting normal forms of types. Thus, the inferences on IFO$^+$ will also be based on a set of rewrite rules. These rules will work on terms of the form

↑ ({ type }) ({ structure-declaration }) ({ role-declaration })

which may be regarded as a slight extension of type declarations in IFO$^+$ allowing multiple structure-declarations to be used. We will refer to the three components of such a term as the *IsA-part*, the *structure-part* and the *role-part* respectively. Furthermore, we use the symbol ^ to denote the concatenation of list, e.g. $struct_1 \hat{} struct_2$ will denote the concatenation of two structure-parts. For rules we use the notation $\frac{T}{T'}$ which means that the term T can be transferred into the term T'.

3.1 Normalization Rules

The first group of rules is intended to gather all the information corresponding to structure- and role-inheritance by propagating upwards along defined supertype-links. The easiest case concerns a type occurring directly in the IsA-part of a given type T. This case is handeled by rule 1.

rule 1
$$\frac{\uparrow(\ldots\uparrow IsA'\,struct'\,roles'\ \ldots)\,struct\,roles}{\uparrow IsA'\hat{}(\ldots\overbrace{\uparrow IsA'\,struct'\,roles'}^{\text{omit}}\ \ldots)\,struct'\hat{}struct\ \ roles'\hat{}roles}$$

Rule 1 can be viewed as a direct extension of a rule usually used with TSLs. If there were no structure-part, then rule 1 would be a direct reformulation of such a rule.

The next three rules concern a type occuring within the IsA-part of some type T' occurring in some structure in the structure-part of the given type T. This case is more tricky, since we have to take into account types that are not present in the schema. Rule 2 handles the case of a tuple-structure:

rule 2
$$\frac{\uparrow IsA\,(\ldots[\,L_1\ \uparrow IsA'\,struct'\,roles'\,L_2\,]\ \ldots)\,roles}{\uparrow(\uparrow()([\,L_1\,T\,L_2\,])\,roles'')\hat{}IsA\,(\ldots[\,L_1\ \uparrow(T)\hat{}IsA'\,struct'\,roles'\,L_2\,]\ \ldots)\,roles}$$

where $roles''$ is the role-part of a type T' in S with the structure $[\,L_1\,T\,L_2\,]$ provided such a T' exists. If not $roles''$ is empty.

The rules 3 and 4 are built analogously using the disjoint union or the set constructor respectively.

3.2 Reduction Rules

The second group of rules contains reduction rules that are intended to remove redundancies from terms. The successive application of normalization rules "creates" new components in the structure- and in the role-part of a term. However, in the resulting term redundant information may occur, e.g. a role-name r may occur twice say as $r : T_1$ and $r : T_2$. If T_2 is a supertype of T_1, the component $r : T_2$ is redundant and can therefore be omitted. This is done by the following rule:

rule 5
$$\frac{\uparrow()\,struct\,(\ldots r:T\ \ldots\ r:T'\ \ldots)}{\uparrow()\,struct\,(\ldots r:T''\ \ldots\overbrace{r:T'}^{\text{omit}}\ \ldots)}$$

if there exists a common subtype T'' of T and T' in S.

Note that rule 5 is a rule usually used in TSLs if we neglect the structure-part.

A similar situation may occur with the structure-part, e.g. there may be two tuple-structures $[T_1 \ldots T_n]$ and $[T'_1 \ldots T'_n]$ of the same length such that T'_i is a supertype of T_i for each i, hence $[T'_1 \ldots T'_n]$ is redundant. Rule 6 handles this case of omitting redundant tuple-structures. Analogously there are two further rules that apply to redundant disjoint union and redundant set structures respectively. For the sake of simplicity we assume the IsA-part is these rules to be empty which can always be achieved by applying normalization rules.

rule 6
$$\frac{\uparrow()\,(\ldots[\,T_1\ \ldots\ T_n\,]\ \ldots[\,T'_1\ \ldots\ T'_n\,]\ \ldots)\,roles}{\uparrow()\,(\ldots[\,T_1\ \ldots\ T_n\,]\ \ldots\overbrace{[\,T'_1\ \ldots\ T'_n\,]}^{\text{omit}}\ \ldots)\,roles}$$

if for each $i = 1, \ldots, n$ the type T_i is a subtype of T'_i.

Note that reduction rules either work only on the structure-part or the role-part.

3.3 Capacity Preserving Rules

The third group of rules comprises the capacity preserving restructuring rules known from IFO [2]. Thus, these rules work only on the structure-part of a term. The first four rules are used to flatten nested tuples (rules 9 and 10) and analogously for nested unions (rules 11 and 12).

rule 9 $$\frac{\uparrow IsA\,(\ldots[\,L_1\,[\,L_2\,]\,L_3\,]\,\ldots)\,roles}{\uparrow IsA\,(\ldots[\,L_1\,L_2\,L_3\,]\,\ldots)\,roles}$$

rule 10 $$\frac{\uparrow IsA\,(\ldots[\,T\,]\ldots)\,roles}{\uparrow IsA\,(\ldots T\,\ldots)\,roles}$$

A more complex situation arises if we regard two different nested constructors. We consider the distribution of tuples over unions in rule 13. Moreover, a set of a union may be replaced by a tuple of sets which corresponds to some kind of "sorting".

rule 13

$$\frac{\uparrow IsA\,(\ldots[\,T_1\,\ldots T_{i-1}\,<\,S_1\,\ldots\,S_m\,>\,T_{i+1}\,\ldots\,T_n\,]\,\ldots)\,roles}{\uparrow IsA\,(\ldots<[\,T_1\,\ldots T_{i-1}\,S_1\,T_{i+1}\,\ldots\,T_n\,]\ldots[\,T_1\,\ldots T_{i-1}\,S_m\,T_{i+1}\,\ldots\,T_n\,]>\,\ldots)\,roles}$$

rule 14 $$\frac{\uparrow IsA\,(\ldots\{<\,T_1\,\ldots\,T_n\,>\}\,\ldots)\,roles}{\uparrow IsA\,(\ldots[\,\{\,T_1\}\,\ldots\,\{\,T_n\,\}\,]\,\ldots)\,roles}$$

4 Structural and Absolute Equivalence

So far we have introduced a new TSL called IFO$^+$ that extends at the same time a basic TSL and the semantic data model IFO. Classification in TSLs and restructuring in SDMs are rewrite based inference mechanisms. We extended the usual rules for these tasks and defined a set of rewrite rules over IFO$^+$-schemata. It can be shown that this set of rules is *terminating* and *confluent* [8], hence satisfies the Church-Rosser property, i.e. for each type T in a schema S there exists a unique normal form $\mathcal{N}^*(T)$. This normal form can be derived from T by successive application of the rewrite rules until no more rules are applicable. We call $\mathcal{N}^*(T)$ the *cp normal form* of T.

Moreover, if we restrict ourselves to normalization and reduction rules as introduced in section 3 the resulting set of rules is also terminating and confluent. Hence there exists another normal form $\mathcal{N}(T)$ corresponding to this restricted set of rules. We call $\mathcal{N}(T)$ the *reduced normal form* of T.

We omit the formal proofs, but due to the fact that the corresponding rule sets for TSLs and IFO satisfy the Church-Rosser property we could make use of the existing proofs. Moreover, a general idea results from the fact that in all our rules we always remove parts of type-declarations and add other parts ranging "higher" in the type hierarchy.

Now we exploit the existence of normal forms in order to determine whether there exists a non-empty model for each type in a schema. Moreover, we are able to characterize equivalent types. For the latter task we introduce formally the notions of *structural* and *absolute* equivalence. The difference between this notion is that absolute equivalence abstracts from the concrete structure of a type and only regards the information capacity.

4.1 Subsumption and Absolute Dominance

The model-theoretic semantics of IFO$^+$ does not tell us whether there exists a model for each type T in a schema S. Therefore, we introduce the notion of a consistent schema.

> S is called *consistent* iff $\epsilon(T) \neq \emptyset$ for all types T in S and all domains D.

In any IFO$^+$-schema we allow subtypes to be defined explicitly. However, due to the introduction of defined types there may exist other subtype relations that can be inferred. The corresponding inference mechanism in TSLs is usually called *classifier*. The classifier detects all explicitly or implicitly specified subtype relations. This is formalized by the notion of *subsumption*.

> A type T *subsumes* another type T' in a schema S iff $\epsilon(T') \subseteq \epsilon(T)$ for all possible domains D.

Note that the usual definition of subsumption in TSLs [10] carries over directly to IFO$'$-schemata. However, subsumption is always structure preserving, e.g. it never relates a tuple of tuples with a "flattened" tuple, although this would not destroy the represented information. Therefore, in SDMs another implicitly defined relation between types has been studied: *absolute dominance* [2]. The rough idea behind absolute dominance is that type T should dominate type T' iff all information that is representable by a value in $\epsilon(T')$ can also be represented in $\epsilon(T)$. In contrast to subsumption absolute dominance abstracts from the structural aspect of types. Extending the definition given in [2] for the case of IFO to IFO$^+$ is straightforward:

> T_1 *dominates* T_2 *absolutely* iff for all domains D there exists some natural number k such that $\mid \epsilon_X(T_2) \mid \leq \mid \epsilon_X(T_1) \mid$ for all finite sets X with $\mid X \cap \epsilon(B) \mid \geq k$ for all types $B \in S$ with $\epsilon(B) \subseteq D_0$ occurring in T_1 or T_2, where
>
> $$\epsilon_X(T) = \{d \in \epsilon(T) \mid \forall x \in D_0.(x \text{ occurs in } d \Rightarrow x \in X)\}\,.$$

If T subsumes T' and vice versa, we call T, T' *structurally equivalent*. If T absolutely dominates T' and vice versa, we call T, T' *absolutely equivalent*.

4.2 Normal Form Characterization

We are now prepared to characterize subsumption and absolute dominance in terms of the normal forms. It turns out that we use the reduced normal form $\mathcal{N}(T)$ for subsumption and the cp normal form $\mathcal{N}^*(T)$ for absolute dominance. Both normal forms have the following structure:

> $\uparrow\ ()\,(S_1 \ldots S_n)\,(r_1 : T_1 \ldots r_m : T_m)$, where each S_i has the form $[T'_{i1} \ldots T'_{ik_i}]$, $< T'_{i1} \ldots T'_{ik_i} >$ or $\{T'_i\}$. Moreover, each T_l and each T'_{il} is a structureless type-name (but there exist also normal form expressions for types with these names).

Then S is consistent iff for each type T in S the number n of structure expressions in the cp normal form $\mathcal{N}^*(T)$ is 1 and each role-name r_l occurs at most once in $\mathcal{N}^*(T)$.

This is due to the fact that our reduction rules simplify all compatible structure and role expression. Thus, all remaining expressions are either incompatible in which case there is no model for T or incomparable. We omit the formal proof.

Hence for a consistent schema S we have

$$\mathcal{N}(T) \;=\; \uparrow()\,\mathcal{S}(T)\,(r_1 : T_1 \;\ldots\; r_m : T_m) \quad \text{and}$$

$$\mathcal{N}^*(T) \;=\; \uparrow()\,\mathcal{S}^*(T)\,(r_1 : T_1 \;\ldots\; r_m : T_m)$$

with pairwise different role-names r_i. In defining the reduction rules we already used a partial order on structure expressions such as $\mathcal{S}(T)$.

We have $[T_1 \ldots T_n] \trianglelefteq [T'_1 \ldots T'_m]$ iff $n = m$ and T'_i subsumes T_i for all i.

The definition is analogous for the disjoint union and for the set constructor.

From TSLs [10] we know that the normal forms of the types are not sufficient for the characterization of subsumption. Indeed, the rules we introduced so far do not take much care about the distinction between primitive and defined types. This forces us to take also the primitive supertypes into consideration. Hence the following definition of the *primitive hull* $Prim(T)$ of a type T in a schema S. Each $Prim(T)$ is a set of structureless type-names.

- If $T :< \uparrow (T_1 \ldots T_n)\, roles$ is a primitive type declaration, then we define

$$Prim(T) \;=\; \{T\} \cup Prim(T_1) \cup \ldots \cup Prim(T_n)\,.$$

- If $T :=\uparrow (T_1 \ldots T_n)\, struct\, roles$ is a defined type declaration, we define

$$Prim(T) \;=\; \{T\} \cup Prim(struct) \cup Prim(T_1) \cup \ldots \cup Prim(T_n)\,,$$

  where $Prim(struct)$ is defined as follows:

  - $Prim([T_1 \ldots T_n]) \;=\; \{[T'_1 \ldots T'_n] \mid T'_i \in Prim(T_i)\}\,,$
  - $Prim(< T_1 \ldots T_n >) \;=\; \{< T'_1 \ldots T'_n > \mid T'_i \in Prim(T_i)\}$ and
  - $Prim(\{T_1\}) \;=\; \{\,\{T'\} \mid T' \in Prim(T)\,\}\,.$

Now we are able to characterize subsumption in terms of the reduced normal forms and primitive hulls provided the underlying schema has been shown to be consistent. Then T_1 subsumes T_2 iff the following three conditions are satisfied:

(i) $Prim(T_2) \subseteq Prim(T_1)$,

(ii) $\mathcal{S}(T_2) \trianglelefteq \mathcal{S}(T_1)$ and

(iii) each role-name r occurring in the role-declaration part of $\mathcal{N}(T_1)$ (say $(r : T)$) also occurs in the role-declaration part of $\mathcal{N}(T_2)$ (say $(r : T')$) such that T subsumes T'.

A similar characterization holds for absolute dominance. The only difference is that we use $\mathcal{N}^*(T)$ instead of $\mathcal{N}(T)$. Thus, T_1 dominates T_2 absolutely iff the following three conditions are satisfied:

(i) $Prim(T_2) \subseteq Prim(T_1)$,

(ii) $\mathcal{S}(T_2) \trianglelefteq \mathcal{S}(T_1)$ and

(iii) each role-name r occurring in the role-declaration part of $\mathcal{N}(T_1)$ (say $(r : T)$) also occurs in the role-declaration part of $\mathcal{N}(T_2)$ (say $(r : T')$) such that T dominates T' absolutely.

These two characterizations are the natural extensions of subsumption in TSLs [10] and absolute dominance in IFO [2].

4.3 A sample Application of Restructuring

We look again at the formely defined example of section 2.3.
We derive:

- Owner subsumes Boat-Owner
- Vehicles subsumes Boats

The reduced normalform for this example is illustrated in figure 2.

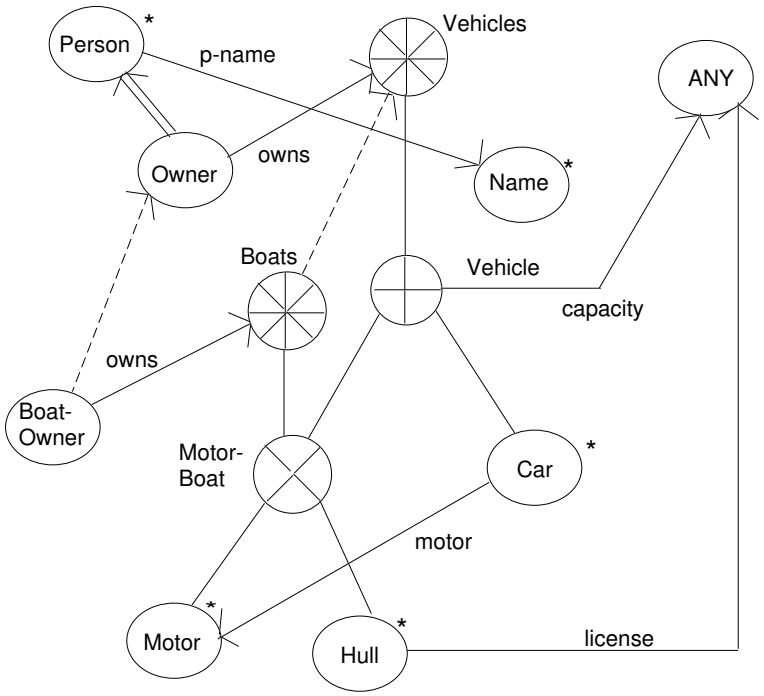

Figure 2: Normalform

The cp normalform is shown in figure 3. Note, that a new type, named CARS is introduced.

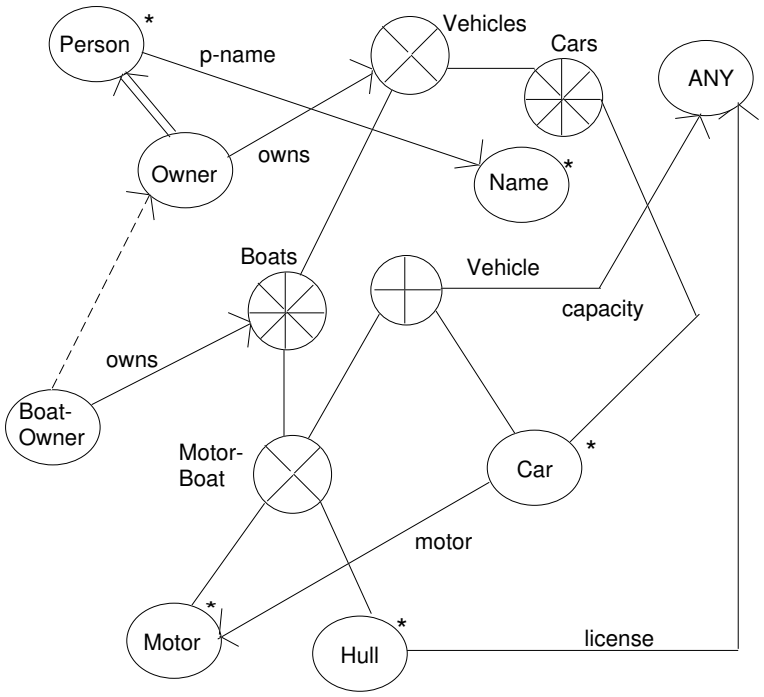

Figure 3: Normalform*

5 Conclusion

We presented an extended TSL called IFO$^+$ which takes a basic term subsumption language and extends it by introducing type constructors as they are used in semantic data models such as IFO. Both IFO- and TSL-schemata are representable in IFO$^+$.

So far our model is restricted to basic features. We did not take into account multi-valued roles nor cardinality constraints that are common in TSLs. However, we already have a finite set constructor. Thus, an extension of IFO$^+$ into this direction is not very difficult. Moreover, we did not handle role hierarchies, but adding them seems also not to be too hard a task.

We have shown that on the basis of IFO$^+$ classification in TSLs and restructuring in SDMs can be unified using a term rewriting approach. The described set of rules turns out to satisfy the Church-Rosser property. Then subsumption in IFO$^+$ which generalizes subsumption in TSLs can be characterized in terms of a normal form. This normal form arises from the application of a subset of the IFO$^+$ rewrite rules. Moreover, the normal form arising from the application of the whole rule set can be used to characterize absolute dominance in IFO$^+$ which generalizes the corresponding notion in IFO.

The impact of IFO$^+$ on conceptual modelling is twofold. First of all IFO$^+$ can be used instead of IFO which has already been proven to be usefull for this task, but now we are able to apply a classifier to a conceptual schema. Hence implicit relationships can be detected. Moreover, restructuring of IFO-schemata is preserved within IFO$^+$. Restructuring has been introduced to support the change or integration of schemata. Hence, this advantage of IFO carries over to IFO$^+$.

The extension of a TSL by type constructors can be regarded as an impact of IFO$^+$ on

knowledge representation, since it now allows complex structured objects to be modelled without loosing the advantage of having a classifier. It is an open question whether this approach can be extended to more sophisticated type constructors such as polymorphic functions [3].

Another open question concerns the management of change propagation. TSLs have been shown to be a usefull basis for this task [13], but the question is how an extension in IFO$^+$ could look like.

References

[1] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Computing Surveys*, 12(4):525–565, December 1987.

[2] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 1988.

[3] A. Borgida. Frames as terms and types (Initial Investigations). Technical report, Rutgers University, 1991.

[4] A. Borgida, R.J. Brachman, D.L. McGuiness, and L.A. Resnick. CLASSIC: A structural data model for objects. In *ACM-SIGMOD Intern. Conf. on the Management of Data*, pages 58–67, Portland, Oregon, 1989. ACM Press.

[5] R. Brachman. On the epistemological status of semantic networks. In N.V. Findler, editor, *Associative Networks*. Academic Press, New York, 1979.

[6] R. Brachman, R. Fikes, and H. Levesque. KRYPTON: A functional approach to knowledge representation. *IEEE Computer*, 16(11), 1983.

[7] R. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2), 1985.

[8] G. Huet. Confluent reductions: Abstract properties and application to term rewriting systems. *Journal of the ACM*, 27(4), 1980.

[9] R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3), September 1987.

[10] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*. Springer LNAI, New York, Berlin, 1990.

[11] B. Nebel and K. von Luck. Hybrid reasoning in BACK. In *Proc. of ISMIS-88*. Elsevier, 1988.

[12] B. Piza. Termsubsumption mit Typkonstruktoren (in German). *M.Sc. thesis, University of Hamburg*, April 1992.

[13] K.-D. Schewe. Variant construction using constraint propagation techniques over semantic networks. In *Proc. 5th Austrian AI Conference*, pages 188–197, Innsbruck, Austria, 1989. Springer IFB 208.