

# Towards a Semantic Web Modeling Language

Christoph Wernhard

Persist AG, Rheinstr. 7c, 14513 Teltow  
Tel: 03328/3477-0, Email: wernhard@persistag.com

## 1 Introduction

**Schemas in the Semantic Web** The Semantic Web [2] requires that reference models, schemas specifying structuring and access methods of information, can be shared by consumers and providers of information. Every net user should be able to publish schemas: entirely new models, sophisticated refinements of given models and also simplifications of given models which are adapted to an application field or user community.

A language to express such models in the Semantic Web has a number of further functions: it should provide human understandable structuring of knowledge; it should allow to define mappings between different schemas related to a common subject; standard techniques from knowledge based modeling such as object orientation, constraints and defaults should be incorporated; defining language extensions corresponding to particular knowledge representation approaches and mappings to different other programming and representation languages should be facilitated.

**Schemas, Models, Interfaces, Types** Different schema level representation formalisms, like relational database schemas, object-oriented classes and object-oriented interfaces emphasize different aspects such as data structuring, code sharing, model based representation, and interfaces between providers and customers of information.

Our approach starts from considering types in the sense of interfaces only. As familiar from the Java programming language, types (interfaces) are distinguished from classes. With types only method signatures and no implementation or concrete data structuring information is associated. Very flexible ways of combination, such as multiple inheritance, can be specified for types in simple ways. Since agreement on supplied and required interfaces is all that is needed for the exchange of data in a distributed environment, types already provide the glue for many useful applications.

**A Type Language** So we outline here a small Web embedded language for specifying types and reasoning about them. Its main features are:

- From primitive types, that require from their instances just that they implement a single method, complex types can be constructed by a few operators, such as as a type intersection.<sup>1</sup> This composition of type definitions maps to the composition of fragments of type definitions in different Web documents.
- Types are compared by *structural equivalence* [3] and not based on their names. This facilitates combination of independently developed models and implicit association of types with given instances.
- The hyperlinking of expressions of the modeling language can be understood in terms of defining equations of a rewriting system. The left side of such a rewrite rule is an URI, the right side an expression. Hyperlink dereferencing corresponds to a rewrite step with such a rule.

Within an open system like the Web, it can be a problem to decide whether the information relevant for a certain task has been gathered completely. The expression rewriting approach ensures that all information about a type can be determined by transitively following hyperlinks.

**Abstract Syntax** The popularity of XML based formats for representing structured information appears from a certain point of view as as recognition of the importance of abstract syntax. This is exemplified in particular by approaches to provide XML data bindings for programming languages (e.g. [6]). Unfortunately the “concrete syntax of the abstract syntax” provided by XML and its document type definitions is hardly readable for humans in areas outside classical document processing. We therefore use functional notation to describe abstract syntax. It should be easily mappable into XML, ISO Prolog, Common Lisp, a variant of Corba IDL and a meta object protocol.

---

<sup>1</sup>Such a composition of types is used in the programming language Forsythe [7]

## 2 Type System

### 2.1 Primitive Types

A primitive type is a type that has just a single message.<sup>2</sup> In the following example a type `colored_object` is defined, that has the message `get_color`, which takes no arguments and returns an object of type `string`.

```
colored_object =  
  get_color : msg(string)
```

The general syntax is:

```
primitive_type ::= name : msg( $\overline{type}$ , type) | void
```

“:” is a binary constructor function for primitive types. *name* is a string that is used as message name.  $N:\text{msg}(\overline{T}_a, T_v)$  denotes the type which has just the message *N* on the sequence of arguments of types  $\overline{T}_a$  and with value type  $T_v$ . Special cases are types which have just a no-argument message:  $N:\text{msg}(T_v)$ , and types which have just a message that doesn't return a value:  $N:\text{msg}(\overline{T}_a, \text{void})$ .

$N:\text{msg}(T_{a1} \dots T_{a1_n}, T_{v1})$  is a subtype of  $N:\text{msg}(T_{a2_1} \dots T_{a2_n}, T_{v2})$  if  $T_{v1}$  is a subtype of  $T_{v2}$  and for  $i = 0 \dots n$   $T_{a2_i}$  is a subtype of  $T_{a1_i}$ . Notice that the subtype relation is inverted (contravariant [3]) for the argument types. A field writer for example has the accepted value type of the field in the contravariant argument type position of `msg` and result type `void`.

A semantics of these primitive types can be expressed in first order logic with the help of two predicates `in` and `send`.  $\text{in}(X, T)$  denotes that object *X* has type *T*. The role of this predicate is similar to  $\in$  in the common axiomatizations of set operators.  $\text{send}(N, X, A, V)$  states that *V* is the result of sending message *N* with argument *A* to object *X*. A message is represented by  $\text{send}(N, X, A, V)$  as a partial function, that (for given *N* and *X*) is defined on each argument in its domain *A*.

$$\forall X, N, T_a, T_v \text{ in}(X, N : \text{msg}(T_a, T_v)) \leftrightarrow \forall A \text{ in}(A, T_a) \rightarrow \exists V \text{ send}(N, X, A, V) \wedge \text{in}(V, T_v)$$

### 2.2 Intersection Types

The binary constructor `&` builds the intersection type of two types. An instance has the intersection type of two types if it has both of them.

```
composed_type ::= type & type
```

Intersection of types can be used to construct a type supporting different messages directly from primitive message types. For example the type `point` has two messages, `get_x` and `get_y` which return objects of type `int`:

```
point =  
  get_x : msg(int) & get_y : msg(int)
```

Intersection of types can also be used to construct a type in the manner of multiple inheritance from given combined types and given additional primitive types. In the following example `colored_point` extends `point` by the `get_color` message:

```
colored_point =  
  point & get_color : msg(string)
```

`colored_point` is a subtype of `point`, since it imposes a stronger condition on its instances. Through structural type equivalence it is also a subtype of `colored_object` as defined in the previous subsection. Furthermore, through structural equivalence the following two expressions denote types equivalent to `colored_point`:

```
point & colored_object  
  
get_x : msg(int) & get_y : msg(int) & get_color : msg(string)
```

---

<sup>2</sup>A *type has a message* means that the type requires its instances to implement this message.

## 2.3 Further Combined Types

**Simulating By-Name Equivalence** To simulate by-name type equivalence, a further primitive type constructor, `tag`, can be used:

$$\text{primitive\_type} ::= \text{name} : \text{msg}(\overline{\text{type}}, \text{type}) \mid \text{name} : \text{tag} \mid \text{void}$$

Two types, which otherwise implement the same messages, can be distinguished by the tags they have. These tags are not linked with any further constraints, although a more specific type-system might associate constraints with them, for example to ensure that comparison of tags is sufficient for type checking.

**Union Types** An object has the union type of two types, if it has one or the other of them. This rather general notion can be made more specific in different ways: 1. the union type should be the type having just those methods supported by both types, i.e. the union constructs the least common supertype; 2. the object must have one of both types; 3. the union type is understood as disjoint union (sum type), which can be simulated by (2.) along with `tag` primitive types; or 4. in the general sense of classical disjunction, as providing just information that the object has one type or the other type. It is currently not obvious which of these notions is best suited for a Semantic Web modeling language.

**Recursive Types** Defining equations, as in the examples above allow to specify recursive types. Algorithmic characterizations of the subtype relationship and equivalence between recursive types, although not in the presence of intersection and union types, are given by Amadio and Cardelli [1] and Jim and Palsberg [4].

**Polymorphic Types** A polymorphic type can be specified by using type variables in the defining equation. All the variables in an equation must appear on the left side.

## 3 Web Embedding

The following table summarizes the syntax of our type language as outlined above, and adds syntactic elements for definitions and hyperlink references:

$$\begin{aligned} \text{definition} & ::= \text{reference} \longrightarrow \text{type} \\ \text{reference} & ::= \text{ref}(\text{URI}) \\ \text{type} & ::= \text{reference} \mid \text{primitive\_type} \mid \text{composed\_type} \\ \text{primitive\_type} & ::= \text{name} : \text{msg}(\overline{\text{type}}, \text{type}) \mid \text{name} : \text{tag} \mid \text{void} \\ \text{composed\_type} & ::= \text{type} \ \& \ \text{type} \end{aligned}$$

Type defining equations can now be considered as rewrite rules, for example:

$$\begin{aligned} \text{ref}(\text{"http://test/colored\_point"}) & \longrightarrow \\ & \text{ref}(\text{"http://ml/examples/point"}) \ \& \\ & \text{get\_color} : \text{msg}(\text{ref}(\text{"http://ml/base/string"})) \end{aligned}$$

Notice that, while the other functors in the type language are data term constructors, `ref` is a function.

In practice the left sides of these equations might be implicit through the URI by which they are accessed. The association of lookups of hyperlinked expressions with rewrite rules is obviously trivial, but it makes the notion of expression fragments containing inline-able hyperlinks to other expression fragments more precise. Such precision is a prerequisite for more complex issues, such as Web-distributed query optimization or the relationship of meta-information to proofs outlined in section 5.

For polymorphic types the *reference* and *primitive\_type* elements can be modified to include type variables:

$$\begin{aligned} \text{definition} & ::= \text{ref}(\text{URI}, \overline{\text{variable}}) \longrightarrow \text{type} \\ \text{reference} & ::= \text{ref}(\text{URI}, \overline{(\text{type} \mid \text{variable})}) \\ \text{primitive\_type} & ::= \text{name} : \text{msg}(\overline{(\text{type} \mid \text{variable})}, (\text{type} \mid \text{variable})) \mid \text{name} : \text{tag} \mid \text{void} \mid \text{any} \end{aligned}$$

## 4 Processing

On the operational level there are three main tasks:

- Proving the subtype relationship for two given types.
- Proving type equivalence. This task can be reduced, with some possible loss of efficiency, to proving the subtype relationship in both directions.
- Proving the instance-of relationship.

While first two tasks can be processed within the type language alone, the third connects the instance level.

The common way to specify type systems is algorithmically in form of inference rules [3]. This has some drawbacks: 1. Algorithmic specification is not a semantics. 2. It is sometimes not evident which typing rules actually have to be included since for different type constructors often rules expressing their behavior in combination are needed. 3. There might be better algorithms than naive interpretation of a set of typing rules. Well-known relevant processing techniques from the area of automated deduction do not as easily transfer to type processing as might be possible.

While knowing that it is difficult, we suggest another approach in which the semantics of a type system is first specified in a general logic language. The processing algorithm could then be described or even automatically generated as a specialized prover for that general logic language.

## 5 Meta-Information – Derivations

The rewriting model suggests to view URI resolving as an equational proof task. Some theorem proving programs record the steps connecting axioms and goal by constructing a proof object during search. This is typically a graph which connects the names and parameters of the applied deduction rules as nodes. The handling of meta information associated with Web documents can be considered analogously: A Web processor can return a derivation object representing the meta information encountered while resolving references.

Such a derivation node for a document (rule) can contain for example information about:

- Date, size, author.
- Relationship to other documents, e.g. whether this document should be considered a subsequent version of another one, or whether this document should be considered as derived from another one by application of certain operations.
- Information about the current reduction step.
- Information about preceding steps which made the current reduction step possible, e.g. sub-derivations used to generate the right-side before returning it to the client.

The following example rule illustrates this. The operator @ associates derivation information with the rewriting rule. `rewrite` says that the right side is obtained by a rewriting step using this rule (i.e. looking up "`http://test/colored_point`"), `operation` provides additional information about the construction of the returned type object, which is not captured by the structural equivalence, but can be of use in certain contexts, such as tracing back the genesis of a document. `author` is the author of the rewrite rule, responsible for its validity. This property can be used to decide about the trustworthiness of the rule.

```
ref("http://test/colored_point") →
  ref("http://ml/examples/point") &
  get_color:msg(ref("http://ml/base/string"))
@ [rewrite = "http://test/colored_point",
   operation = add_field("http://ml/examples/point", color),
   author = "john@colors.org"]
```

A Web processor combines such nodes (primitive derivations) to a derivation graph (complex derivation). Note that derivations can be checked already during their construction, for example they can be restricted such that

only sites which are trusted in some sense are considered. Furthermore derivations can be simplified and abstracted during construction to contain only information relevant to the ultimate client. A client can submit its requirements on the returned derivation to a server.

The inclusion of derivations extends the “extensional” view of a Web reference — dereferencing as a semantics preserving rewrite step — to an extension-intension pair, that also contains a representation of “intensional” aspects: a proof or derivation, including epistemic information such as authorship.

## 6 Conclusion

We believe that some characteristics of the outlined modeling language are important features for distributed semantic expression in the Web, especially the composition of types from primitive types and the embedding of equational specification into the URI space.

The World Wide Web Consortium suggest with RDF-Schema [5] a rudimentary logic modeling language that uses property-object-value assertions. Such statements are positive assertions about the extension of a property. Negative facts, that delimit the extension, are in Horn clause programming and Datalog [8] implicit by restriction to minimal models with respect to the set of propositions of the program or database. For a logic program or a deductive database, this set of propositions is completely determined. In a distributed setting like the Semantic Web, however, it could be a problem to determine, when the set of propositions that have to be considered for a reasoning task is complete. The use of equational specification seems to avoid this problem, since an equation characterizes positive as well as negative aspects.

There are a number of issues left open for further research: Work out a precise semantic characterization of the type system. Work out an implementation according to the suggestions of section 4. Describe simulation of specific data models as specializations of the type system. Work out instance level languages.

## References

- [1] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575-631, September 1993.
- [2] Tim Berners-Lee, Dan Connolly, Ralph R. Swick: *Web Architecture: Describing and Exchanging Data*, W3C Note 7 June 1999, <http://www.w3.org/1999/06/07-WebData>.
- [3] Luca Cardelli: *Type Systems*. In *Handbook of Computer Science and Engineering*. CRC Press 1997.
- [4] Trevor Jim, Jens Palsberg: *Type inference in systems of recursive types with subtyping*. Manuscript, 2000. Available from <http://www.cs.purdue.edu/homes/palsberg/publications.html>.
- [5] *Resource Description Framework (RDF) Schema Specification*, W3C Proposed Recommendation 03 March 1999, <http://www.w3.org/RDF/>.
- [6] Mark Reinold: *An XML Data-Binding Facility for the Java Platform*. Palo Alto, 1999. Available from <http://java.sun.com/xml/white-papers.html>.
- [7] John C. Reynolds: *Design of the Programming Language Forsythe*, Technical Report CMU-CS-96-146, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1996.
- [8] Jeffrey D. Ullman: *Principles of Database and Knowledge-Base Systems*, Rockville, 1989.