Representing Disjunction and Quantifiers in RDF *

Drew McDermott and Dejing Dou

Yale Computer Science Department
New Haven, CT 06520, USA
{drew.mcdermott,dejing.dou}@yale.edu

Abstract. The advantage of the RDF/DAML+OIL family of languages over ordinary XML is that it is topic-neutral and composable. However, its expressivity is severely limited. This limitation is well known, and the usual remedy is *reification,* in which RDF is used to describe formulas in a richer language. We propose a method for encoding typed predicate calculus using reification, which handles bound variables cleanly and causes the size to increase by only a constant multiple. The method generalizes to virtually any system, a claim which we illustrate by describing our program, PDDAML, which encodes domain specifications in PDDL using our technique. We argue that reification, while logically suspect, is in practice benign because any algorithm capable of doing inferences using logical notations can be easily extended to "unreify" those notations as needed. We also argue that the ability to represent predicate calculus on the semantic web is crucial.

1 Introduction

The "semantic web" really ought to be called the "inferential web." Although people often talk as though RDF [7] and related tools such as DAML+OIL [12] specify the "meaning" of symbols, they really specify relations among symbols that allow inferences to be drawn. RDF can be thought of as a simple logical system in which assertions are made about objects denoted by URIs. By providing a uniform syntax, it allows for different RDF documents to be combined in a straightforward way. This contrasts with XML, in which the allowable contents of an element depend entirely on the meaning of its tag. In RDF, whether an element is a description or a property is easy to determine syntactically, and RDF puts no restrictions at all on what values a property can have. Hence an arbitrary RDF node can occur as the value of a property, and that node can itself have an indefinite amount of further descriptive material. A description in one document can be pointed to from another, without any restrictions on how the vocabularies of the two documents relate.

* This research was supported by the DARPA DAML program. The technical ideas in the paper were arrived in collaboration with Jonathan Borden, Mark Burstein, Doug Smith, and several other contributors to the `www-rdf-logic` mailing list.

I. Horrocks and J. Hendler (Eds.): ISWC 2002, LNCS 2342, pp. 250–263, 2002.

However, RDF does have limitations:

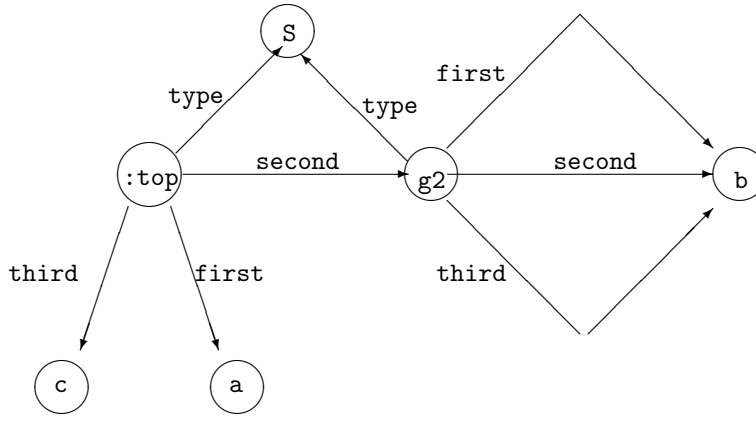
1. Its syntax is a graph structure, reminiscent of semantic networks. Such graph structures have well-known problems with scoping of quantifiers, negations, and disjunctions. RDF solves these problems by disallowing all these constructs. (Anonymous nodes can be considered to be existentially quantified, but the scope of the quantifier is the entire graph [5].)
2. It does not allow arbitrary terms, just bags and sequences.

There are two ways of fixing these bugs: by extending RDF, or by building on top of RDF using some sort of quotation device, such as reification. The former approach seems more natural, but runs into opposition from users and developers who believe that the simplicity of RDF is worth preserving. So in this paper we use the second approach.

Reification is the use of RDF to describe formulas in another language, which we call the *embedded language*. We use the word "another," but of course the language may look a lot like RDF itself. Describing a formula is a matter of providing an RDF graph whose properties correspond to the syntactic relations of the embedded language. For instance, consider a simple language whose grammar is

$S \rightarrow a \mid b \mid c$
$S \rightarrow \langle\ S\ S\ S\ \rangle$

The sentence $\langle\ a\ \langle\ b\ b\ b\ \rangle\ c\ \rangle$ might be described by the RDF graph:



The graph is expressed in "triples" thus:

⟨top rdf:type S⟩
⟨top second g2⟩
⟨top third c⟩
⟨top first a⟩
⟨g2 rdf:type S⟩

⟨g2 first b⟩
⟨g2 second b⟩
⟨g2 third b⟩

Our goal in this paper is to explain how to embed logic in RDF using reification, and to argue that embedding logic this way is a good idea. We will use Lisp-like notation for logical formulas, specifically the typed logic of PDDL [8]. So the formula traditionally written as $\forall x(P(x) \supset Q(x))$ will be written as `(forall (x) (if (P x) (Q x)))`. If type declarations are involved, we declare the types of variables by writing "—*vars*— `-` *type*," as in

```
(forall (x y - animal)
    (if (and (predator x) (bigger x y))
        (fears y x)))
```

2 Our Approach

It is easy to do the wrong thing when embedding logic using reification. For instance, if one thinks of implication as a relationship between two reified formulas, then `(if (if` P Q`)` R`)` becomes a relationship between two objects, the first of which contains the reification of two reifications. As can be seen from the example above, reification blows up the size of anything it is applied to. We can tolerate one layer of this blowup, but any solution that requires reifications of reifications of ... is ruled out.

Fortunately, this will not happen if we think of reifying entire expressions. Again appealing to our simple example, the fact that we have S's inside S's doesn't cause a blowup, because each S becomes a node of type S. Here we use *type* in the technical sense used in [7], and we view it as roughly synonymous with a DAML+OIL *class* [12]. To embed logic, we will simply take `if`, `or`, and other connectives to correspond to classes. So the formula `(if (if` P Q`)` R`)` will get transformed to

⟨top rdf:type drs:If⟩
⟨top drs:antecedent g3⟩
⟨top drs:consequent R⟩
⟨g3 rdf:type drs:If⟩
⟨g3 drs:antecedent P⟩
⟨g3 drs:consequent Q⟩

For conciseness we will use the N3 notation [1,2] for RDF instead of the XML serialization. The result is:

```
:top a drs:If;
     drs:antecedent [a drs:If;
                     drs:antecedent P;
                     drs:consequent Q];
     drs:consequent R.
```

Here we have pretended that P, Q, and R are atomic, but if they aren't we can just continue the process. We use the namespace prefix `drs:` to refer to symbols that are part of our system. For brevity, we will use `drs` as the default namespace, and drop it in what follows.

We extend N3 slightly, in the spirit of the type abbreviations used in [7]. When "`a` T" (N3's notation for `<rdf:type resource="`T`"/>`) occurs as the first element of a node (as it often, by convention, does), we use a little syntactic sugar to write it as T`/`. So the example above could be written:

```
:top If/
     antecedent [If/
                 antecedent P;
                 consequent Q];
     consequent R.
```

When we reach an atomic formula, of the form (*predicate —args—*), we use the basic trick of [7], describing it as having three parts, subject, predicate, and object. So `(loves Fred Sally)` would be represented as

```
[Atomic_formula/ rdf:subj Fred†;
                 rdf:pred loves†;
                 rdf:obj Sally†]
```

where we use daggers to remind you that in practice these terms will be resolved into URIs, or possibly something more complex.

The basic trick won't work unless there are exactly two *args.* In all other cases, we represent the arguments after the first by a *term sequence*, defined as a DAML+OIL list [12] whose elements are all terms.

Terms are defined in the usual way, as constants, variables, or *functional terms*, of the form (*f —args—*). We translate them into an instance of the class `Functional_term`, whose `function` property is $f^\dagger$ and whose `term_args` property is a term sequence. So the logical formula

`(child Bates (mother Bates) (father Bates))`
"Bates is the child of Bates's mother and Bates's father."

would be translated into the RDF

```
[Atomic_formula/
 rdf:subj Bates†;
 rdf:pred child†;
 rdf:obj [Term_seq/
          rdf:_1 [Functional_term/
                  term_function mother†;
                  term_args[Term_seq/
                            rdf:_1 Bates†]];
          rdf:_2 [Functional_term/
                  term_function father†;
                  term_args [Term_seq/
                             rdf:_1 Bates†]]]]
```

Of course, we must also provide an XML serialization of the formula, which is what computers produce for consumption by other computers on the net:

```
<Atomic_formula>
    <rdf:subj> Bates† </rdf:subj>
    <rdf:pred> child† </rdf:pred>
    <rdf:obj>
         <Term_seq>
             <rdf:_1>
                    <Functional_term>
                        <term_function>mother†</term_function>
                        <term_args>
                            <Term_seq>
                                <rdf:_1>Bates†</rdf:_1>
                            </Term_seq>
                        </term_args>
                    </Functional_term>
             </rdf:_1>
             <rdf:_2>
                    <Functional_term>
                        <term_function>father†</term_function>
                        <term_args>
                            <Term_seq>
                                <rdf:_1>Bates†</rdf:_1>
                            </Term_seq>
                        </term_args>
                    </Functional_term>
             </rdf:_2>
         </Term_seq>
    </rdf:obj>
</Atomic_formula>
```

The only part of the translation framework left to explain is how variables are handled. The obvious variable binders (and declarers) are the standard quantifiers `forall` and `exists`, but there are other possibilities, such as the `lambda` expressions allowed by higher-order logic. Here we will focus on the standard quantifiers only, but the idea will work for all variable binders.

When a variable is declared, it is meaningful only within the scope of its quantifier. RDF provides no notion of scoping. A described entity may be given a name by the use of the `ID` attribute in a description of it. If you write `<rdf:Description rdf:ID="x">`, then anywhere else in the file you can refer to the object described by using the attribute `resource="#x"`. Hence IDs behave like existential variables whose scope is the entire graph (or document) they occur in. There is no way to narrow the scope of an ID, thus avoiding name conflicts among variables that happen to have the same name. There is no way to declare an ID to correspond to a universally quantified variable.

The solution is to defer the issue to the quoted level. Name conflicts are avoided by making up a new name for each variable as it is encountered. Variables just become anonymous entities with features such as being universally quantified. So the formula `(forall (x - Person) (moral_agent x))` would become

```
[Forall/
    quantifier_vars
       [variables_list/
           rdf:_1 :var_1];
    body
       [Atomic_formula/
           rdf:subj :var_1;
           rdf:pred moral_agent†]]

:var_1 a Var; name ‘‘x"; type Person†.
```

Here we have made the `rdf:type` property ("`a Var`") of `var_1` explicit, in order to contrast the `rdf:type` and the `type` (i.e., the `drs:type`) properties of a variable, which play two entirely different roles. The former identifies the the variable as a variable, syntactically; the second constrains the values of the variable to be objects of type `Person`.

The collection of classes `Atomic_formula`, `Term_seq`, etc. and their interrelations constitute an *ontology* in the formal sense. A DAML+OIL formalization of that ontology may be found at

`http://www.cs.yale.edu/~dvm/daml/drsonto.daml`.

Let us be perfectly clear about what we are doing here. When one uses reification, one commits to *describing* a formula rather than directly *expressing* it. The description of a formula does not by itself assert that formula. It may be taken as asserting that such a formula exists, but that assertion conveys no useful information, because every syntactically well-formed formula (and perhaps many others) already exist. Hence, reification is useful *only* when used in conjunction with tools that know to treat the description of a formula in a certain context as the assertion of that formula. We will come back to this issue in section 4.

3 Application: Web-PDDL

PDDL, the Planning Domain Definition Language [8], is used to define domains and problems for input to automated planners. It was originally developed as the input language for the semiannual AI Planning Systems (AIPS) Competition, but has gained acceptance as a standard for classical planners generally. PDDL continues to evolve under the guidance of the AIPS Competition Committee. In this paper, we have taken the liberty of evolving it in a slightly different direction, which we will discuss as we go. We will call the result Web-PDDL.[1]

[1] Some of the features we discuss are not yet fully operational, and we will point these out as we go.

Web-PDDL is not just a curiosity as far as the World-Wide Web is concerned. There is intense interest in development of notations for describing web services [3], and PDDL is ideal for that purpose. Its purpose is to describe sets of *actions* an agent can take, and what their *preconditions* and *effects* are. In the Web world, actions comprise the sending and receiving of messages; typical preconditions include knowing data to be included in a message, and typical effects include learning information, putting an order in a "shopping cart," or finalizing a purchase.

Web-PDDL uses the Lisp-like syntax described in section 2, with special extensions for describing actions and declaring the types of symbols. (The resulting enlarged ontology may be found at `http://www.cs.yale.edu/~dvm/daml/pddlonto.daml`) Type declarations, actions, and axioms are organized into *domains*. A domain can inherit the contents of other domains, so they can be mixed and matched [8]. Here is an example:

```
(define (domain www-agents)
  (:extends (uri ``http://www.yale.edu/domains/knowing")
          (uri ``http://www.yale.edu/domains/regression-planning")
          (uri ``http://www.yale.edu/domains/commerce"))
  (:requirements :existential-preconditions :conditional-effects)
  (:types Message - Obj Message-id - String)

  (:functions (price-quote ?m - Money)
              (query-in-stock ?pid - Product-id)
              (reply-in-stock ?b - Boolean)
              - Message)

  (:predicates (web-agent ?x - Agent)
          (reply-pending a - Agent  id - Message-id msg - Message)
          (message-exchange ?interlocutor - Agent
                            ?sent ?received - Message
                            ?eff - Prop)
          (expected-reply a - Agent sent expect-back - Message))

  (:axiom
      :vars (?agt - Agent ?msg-id - Message-id
             ?sent ?reply - Message)
      :implies (normal-step-value (receive ?agt ?msg-id)
                                  ?reply)
      :context (and (web-agent ?agt)
                    (reply-pending ?agt ?msg-id ?sent)
                    (expected-reply ?agt ?sent ?reply)))

  (:action send
      :parameters (?agt - Agent ?sent - Message)
      :value (?sid - Message-id)
```

```
    :precondition (web-agent ?agt)
    :effect (reply-pending ?agt ?sid ?sent))

(:action receive
    :parameters (?agt - Agent ?sid - Message-id)
    :vars (?sent - Message ?eff - Prop)
    :precondition (and (web-agent ?agt)
                       (reply-pending ?agt ?sid ?sent))
    :value (?received - Message)
    :effect (when (message-exchange ?agt ?sent ?received ?eff)
                  ?eff)))
```

This description, of domain `www-agents`, describes two actions, `send` and `receive`[2] The `www-agents` domain *extends* the domains `knowing`, `regression-planning`, and `commerce`, identified by URIs[3]

In turn, this domain can be used as a foundation for other, more specific domains about the content of particular messages. Symbols such as `Agent` (from the `commerce` domain, and `know-val` from the `knowing` domain, can be freely inherited here and in descendent domains. In other words, a domain closely resembles a DAML+OIL *ontology*, a resemblance we shall return to below.

The task at hand is to translate PDDL domain definitions into RDF. Many of the subexpressions of a domain definition are logical formulas of the sort we discussed in section 2, but the top level of a domain includes idiosyncratic constructs such as `(:action` *name* `...)`. But these new constructs present no particular problem, because we can use the same bag of tricks as in section 2, introducing `rdf:type`s such as `pddl:Action`. For example, the embedding of the `send` action into RDF is

```
:send pddl:Action/
      pddl:parameters [Param_seq/ rdf:_1 :ag20; rdf:_2 :me21];
      pddl:value [Param_seq/ rdf:_1 :me22];
      pddl:precondition [Atomic_formula/
                            rdf:subj :ag20;
                            rdf:pred :web-agent;
                            rdf:obj drs:empty];
      pddl:effect [Atomic_formula/
                       rdf:subj :ag20;
                       rdf:pred :reply-pending;
                       rdf:obj [Term-seq/
```

[2] The `:functions` field of the domain definition is an extension of PDDL 1.0, but it will be incorporated into the PDDL2.1 the next official version. The `:value` field of `:action` definitions is our own extension, which we hope will eventually become an official part of PDDL.

[3] For testing purposes, we currently load all such domains in advance, and do not have the program actually visit remote sites for domains such as ‘‘`http://www.yale.edu/domains/commerce`".

```
                                        rdf:_1 :me22;
                                        rdf:_1 :me11]].
  :ag20 Param/ name ‘‘?agt"; type com:Agent .
  :me21 Param/ name ‘‘?sent"; type com:Message .
  :me22 Param/ name ‘‘?sid"; type com:Message-id .
```

(In this formula, the presence of `drs:empty` as the object ("`rdf:obj`") of the predicate `:web-agent` indicates that it is unary.)

The translation from PDDL to RDF is straightforward. We treat PDDL expressions as trees in the usual Lisp way. The first element of each expression gives its basic syntactic type. It is either a PDDL reserved word (such as `:precondition`), a connective (such as `and` or `when`), a predicate (such as `reply-pending`), or a function (such as `price-quote`). Connectives and functions are handled by a recursive walk through their arguments. A predicate is the beginning of an atomic formula, which is reified into four triples (for `rdf:type`, `rdf:subj`, `rdf:pred` and `rdf:obj`) as exemplified above, the only complexity being how the object is handled. In any case, because the subject and object may be arbitrary terms, the recursive walk must continue through them.

Each reserved word has its own special handler. For example, the word `:requirements` must be followed by a list of "requirement" names (we discuss the purpose of these in section 4). The handler for `:requirements` creates a `rdf:Bag` whose elements are strings corresponding to the requirement names, and makes it the value of the `requirements` property of the domain being constructed.

The output of this process is a set of triples, as described in [7], such as

```
:www-agents requirements :bag309 .
```

which are equivalent to an RDF graph.

It is important to keep track of multiple pointers to a given node of that graph. For identifiers like `web-agent`, it is obvious that any occurrence of that identifier must be translated into the same internal node (which we give the name `:web-agent` in N3, or `resource="#web-agent"` in the XML version). For variables we have to use special internal names to avoid scope ambiguities, as explained in section 2. Every time we create a graph node, we enter it into a hash table to ensure that we can find it the next time we need a reference to it.

Having produced the set of triples, we print out its XML serialization, in as readable a format as we can. Space does not allow us to show that serialization here; we will continue to use N3 to show RDF encodings.

Wherever possible, the Web-PDDL translator outputs its content in a way compatible with DAML+OIL. As we said above, a domain resembles an ontology, so in fact we output the standard DAML+OIL ontology boilerplate as the top level of our domain representation:

```
<rdf:RDF
    xmlns:rdf = ‘‘http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs ="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:daml ="http://www.daml.org/2001/03/daml+oil.daml#"
    xmlns:xmls ="http://www.w3.org/2000/10/XMLSchema#"
    xmlns ="">

    <daml:Ontology rdf:about="">
        <rdfs:label>www-agents</rdfs:label>
        <daml:imports rdf:resource="http://www.daml.org/2001/03/daml+oil.daml"/>
        <daml:imports rdf:resource="http://www.yale.edu/domains/drsonto"/>
        <daml:imports rdf:resource="http://www.yale.edu/domains/pddlonto"/>
        <daml:imports rdf:resource="http://www.yale.edu/domains/knowing"/>
        <daml:imports rdf:resource="http://www.yale.edu/domains/regression-planning"/>
        <daml:imports rdf:resource="http://www.yale.edu/domains/commerce"/>
        Insert domain content here
    </daml:Ontology>
    </rdf:RDF>
```

Types in Web-PDDL resemble classes in DAML. In fact, types are simpler than classes. The relationships between types are declared once and for all, and two atomic-named types are disjoint unless one is a subtype of the other. Because they are simpler, PDDL types can simply *become* DAML classes. For the `www-agents` domain, we output[4]:

```
<daml:Class rdf:ID="Message"/>
<daml:Class rdf:ID="Message-id">
    <rdfs:subClassOf rdf:resource="#String"/>
    <daml:disjointWith rdf:resource="#Message"/>
</daml:Class>
```

Our translator can also translate back from a DAML+OIL ontology to Web-PDDL. It checks to see if the ontology imports the Web-PDDL ontology (‘‘`http://www.yale.edu/domains/pddlonto`"), and if so it assumes that an occurrence of a description of a formula is to be taken as an occurrence of the formula itself. It also assumes that the classes declared in the ontology can be turned into Web-PDDL types.

Reconstructing the Lisp-like syntax of formula descriptions is straightforward. The translator parses XML into RDF triples. The triples are isomorphic to a Lisp list structure, with IDs playing the role of pointers, so it is easy to walk through the triples rebuilding the PDDL formulas. Of course, we have to undo the tricks we used for the arguments to functions and non-binary predicates.

We have made our program, called PDDAML, available on the web at `ftp://ftp.cs.yale.edu/pub/mcdermott/daml_pddl_translation_doc.tar.gz`, and also from the `daml.org` tools library (`http://www.daml.org/tools/pddl2daml`). The program is written in Java. The internal object model for RDF graphs is based on that of Jena (an experimental RDF parser written by Brian McBride, and available at `http://www.hpl.hp.co.uk/people/bwm/rdf`), although we made several adjustments to it. We developed the algorithms for parsing, reifying, unreifying, and printing from scratch. Because PDDL looks like Lisp, and

[4] The current version of our algorithm does not actually map types to classes in this way, but treats types as ordinary PDDL symbols.

is conveniently represented as Lisp list structures internally, we used a simple implementation of Lisp's data types in Java. It may occasionally sound as if we are talking about a Lisp program, but Lisp-o-phobes may reassure themselves that the program is pure Java.

4 Objections and Replies

We have encountered several objections to the proposal we make here:[5]

1. "The fact that RDF can be used to encode the syntax of other, richer languages is already well known. DAML+OIL basically used the same trick to encode a description logic in RDF."
2. "DAML+OIL does 'layering' the right way, so that expressions in the new layer are also expressions in RDF, with the same meaning. The encoding of Web-PDDL lacks this property."
3. "Typed predicate calculus has little in common with RDF (e.g., object-oriented notions like class, subclass, range, and domain are not all directly applicable). There's a good reason why RDF has evolved toward description logics and not toward predicate calculus: theorem proving is a computational quagmire."
4. "A scheme of this sort is worthwhile only if everyone uses it. If another scheme gains popularity, no one will write programs that understand Web-PDDL, and so no one will represent anything in Web-PDDL."

To the first objection the reply is simple: We know of *no* previous system for representing disjunction and bound variables in RDF that does not suffer from exponential blowup. DAML+OIL does *not* use a "trick" similar to ours. In fact, it doesn't have to use any tricks, because RDF is essentially a subset of DAML+OIL. That's why DAML+OIL can *not* handle disjunction or implication, which are simply unrepresentable in RDF, due to the fact that asserting any RDF graph asserts all its subgraphs.

To objection 2 we reply that we have tried wherever possible to use DAML+OIL representations. For instance, we declare Web-PDDL types to be DAML+OIL classes instead of reifying their declarations. But there fewer opportunities for using DAML+OIL-style representations than one might think. A simple fact at the top level of a domain would qualify. For instance, if a domain asserted `(sells wabash.com books)`, that could become a single triple `:wabash.com :sells :books.` instead of a description of an atomic formula. But domains usually assert type declarations and axioms, not atomic formulas.

As far as objection 3 is concerned, we agree that general-purpose theorem proving is unlikely to be a useful technique on the web, and that therefore various special-purpose techniques (such as Horn-clause theorem proving, or inference

[5] Some of these are quotes or paraphrases of comments from anonymous referees who rejected an earlier version of this paper submitted to the World-Wide Web 2002 conference.

using description logics) will play a key role. However, we disagree that the best way to control inference is by limiting the syntax of the representation langugage. A good illustration of this point comes from the field of automated planning. The planning problem is intractable, but that has not prevented the development of algorithms that can produce plans with hundreds of steps, in time measured in seconds, not hours.[6] These algorithms require domain specifications using notations of the complexity of Web-PDDL. One way this complexity is controlled is through a set of "requirement declarations" that allow domain definers to specify exactly which subset of PDDL the domain requires. In the `www-agents` domain we specified:

```
(:requirements :existential-preconditions :conditional-effects)
```

which allows a planner that can't cope with existential preconditions or conditional effects to recognize immediately that the domain is beyond its reach. But when all is said and done, any planner can potentially run forever on a problem that appears to be within its domain; the only way to prevent that is to ration the time it is allowed to take.

The obvious conclusion is that "notation-complexity control" is *not* the responsibility of the designers of RDF. On the other hand, RDF is not likely to evolve into a language with syntax so general that every notation in the world (PDDL, KIF [6], ...) is a subset of it. Hence there will probably always be a need for languages embedded in RDF by reification or some other "quoting" technique.

Achieving interoperability across the web requires managing several notational levels. At the lowest level we have XML, which is rapidly becoming a standard. Above that we have XML-based languages, which supply particular vocabulary items to allow domain-specific structures to be described. For many applications, this is all you need. However, to achieve the sort of self-description the semantic web would be based on, we need languages for describing resources in a way that is neutral and composable. RDF can play that role, but to achieve more expressive power we must go one step further and embed more complex languages in RDF.

All these levels may sound messy, but there are good reasons for each level, and the problem of translating them all into a uniform internal representation is tricky but tractable, as we have demonstrated. The difficult part arises when we run into differences in vocabulary — or "ontology" — among different information sources. That is a subject of ongoing research [11].

Finally, there is objection 4, that if our encoding does not become standard then no one will hear of it again. This does bother us, since it may keep us from becoming rich and famous. The real question, however, is whether something with essentially the same power is going to be necessary. We believe the answer

[6] We hasten to add that most of the domains these algorithms can handle are artificial; planners that can actually find plans in realistic domains such as web services are still in the "laboratory curiosity" stage.

is Yes. If the notation of the future improves on our proposal, or even if it's just inexplicably more popular, we will cheerfully switch to it.

5 Conclusion and Related Work

We have argued that it is possible and useful to embed general logic constructs in RDF and XML. We have demonstrated the possibility by providing a program, PDDAML, that translates between RDF/XML and PDDL, the Planning Domain Definition Language. We are incorporating this in a system for planning interactions with web services [9].

The key technical contribution is a method for representing arbitrary formulas with bound variables as elements of RDF classes. Logical symbols of the embedded language (such as `or` in predicate calculus) are translated into `rdf:type`s, so that `(or` P Q`)` becomes a description of an entity of type `Or` whose arguments are $P*$ and $Q*$, the translations of P and Q. For a fuller technical description see [10]. Similar ideas were proposed in [4]. The N3 notation [2] was originally intended as a concise encoding of RDF graphs, and that is how we have used it here. Lately Berners-Lee has made extensions that go beyond the expressive power of RDF; if these should be incorporated into RDF it would make it easier for us to embed logic in RDF, and reduce the number of places where quotation devices would be necessary.

The ability to represent arbitrary formulas in RDF should free us from thinking of the semantic web as graph structures serialized as a nightmarish number of pointy brackets. The semantic web will surely exist as a marketplace of alternative notations, which will show up as alternative quotational ontologies at the notational level, just as alternative ontologies will coexist at the context level. Only a few of the competing notations will survive. We argue that the winners will be those notations that have the following properties:

- *Declarativeness:* Good notations will express truths. Any inference engine with access to their ontologies can make inferences from them.
- *Composability:* An expression from one source can be combined with an expression from another, regardless of whether their designers intended that.
- *Generality:* A notation should be able to express what people want to express.
- *Maintainability:* Information sources will evolve, and they must be comprehensible to their maintainers for this to be so.

If we are right, then notations such as WSDL [3] will wither away in favor of alternatives that *describe* services instead of dictating (to someone with a manual) how to write code to connect to them. Notations such as RDF will have to evolve to allow disjunctions and quantifiers, or quotational devices such as the one we have presented will have to become standard. And, finally, XML serializations should be hidden away from human view lest small children accidentally see them and become frightened. XML is a wonderful way of making data "self describing" to a computer; to a person, it's a way of concealing information. It is especially critical for the semantic web that better surface notations be found.

References

1. Tim Berners-Lee. Primer: Getting into rdf & semantic web using n3, 2000. http://www.w3.org/2000/10/swap/Primer.html.
2. Tim Berners-Lee. Notation 3, 2001. http://www.w3.org/DesignIssues/Notation3.html.
3. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Definition Language (Wsdl) 1.1. Technical report, W3C, 2001. , available at http://www.w3c.org/TR/wsdl.
4. Wolfram Conen, Reinhold Klapsing, and Eckhart Köppen. Rdf M&s revisited: from reification to nesting, from containers to lists, from dialect to pure Xml. In *Proc. Semantic Web Working Symposium*, pages 195–208, 2001.
5. Patrick Hayes. Rdf model theory, 2001. W3C Working Draft 25 September 2001. http://www.w3.org/TR/2001/WD-rdf-mt-20010925.
6. KIF. Knowledge Interchange Format: Draft proposed American National Standard (dpans). Technical Report 2/98-004, ANS, 1998. Also at http://logic.stanford.edu/kif/dpans.html.
7. Ora Lassila and Ralph R. Swick. Resource Description Framework (Rdf) Model and Syntax Specification. Technical report, W3C, 1999. , available at http://www.w3c.org/TR/REC-rdf-syntax.
8. Drew McDermott. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science, 1998. (CVC Report 98-003).
9. Drew McDermott. Estimated-regression planning for interactions with web services. In *Proc. AI Planning Systems Conference 2002*, 2002. To appear.
10. Drew McDermott, Jonathan Borden, Mark Burstein, Douglas Smith, and Richard Waldinger. A Proposal for Encoding Logic in Rdf/daml. Technical report, Yale CS, 2001. ftp://ftp.cs.yale.edu/pub/mcdermott/papers/noworry.ps.gz.
11. Drew McDermott, Mark Burstein, and Douglas Smith. Overcoming ontology mismatches in transactions with self-describing agents. In *Proc. Semantic Web Working Symposium*, pages 285–302, 2001.
12. Frank van Harmelen, Peter F. Patel-Schneider, and Ian Horrocks. Reference description of the daml+oil (march 2001) ontology markup language, 2001. Available at http://www.daml.org/2001/03/reference.html.