The Need for Type Theory in Semantic Web Services

Rod Moten
Bloomberg, L.P.
731 Lexington Ave
NY, NY 10022
rmoten@bloomberg.net

1. INTRODUCTION

My name is Rod Moten. I am a senior computer scientist at Bloomberg L.P., a financial market data company. Currently I'm involved in ontology and taxonomy development. Prior to coming to Bloomberg, I was an assistant professor at Colgate University. While on a sabbatical, I visited RPI. At RPI, I conducted research on using type theory to resolve semantic heterogeneity in multi–agent systems [6, 7]. In this work, I showed how type theory could be used to implement a technique for automatically resolving semantic heterogeneity between software agents. The technique is an offshoot of the *proofs–as–programs* technique created in the 1970's [2, 1].

The proofs–as–programs paradigm was developed by researchers in the formal methods community to develop software that was guaranteed to correctly implement its specification. The specifications were stated as logical assertions. If a proof could be constructed from the specification, then a program could be extracted from the proof. This technique utilized an isomorphism between predicate logic and type theory. The isomorphism, called the Curry–Howard isomorphism [3], identifies logical connectives with type constructors, logical formulas with types, and proof rules with type formation rules. Type formation rules are used to prove that a type is well–formed. Researchers were able to create type formations rules so that proving that a type was well–formed required that one construct a term that inhabits the type [8, 9]. Therefore, if one was able to construct a proof of a logical formula, he was able to create a term that inhabits the type. In type theory, terms are computable. Therefore, if a logical formula represented a specification of a software system, creating a proof of the formula would be a program that adhered to the specification.

This technique not only could be used for creating correct programs, but also for analyzing a specification. By attempting to prove the specification, we can understand which definitions, lemmas, and other theorems are needed for the specification to be true. Each of these lemmas and theorems would represent specifications of other modules. The definitions would be preconditions that are needed for software to work properly.

Resolving heterogeneity between independently created Web services requires the same two–level process as well. First there must be some justification that the conceptual meanings that two parties have can be aligned. Afterwards, converters must be created to translate objects between the systems that adhere to the alignment.

We have conducted some preliminary research to utilize the same foundational properties of type theory as that used in the proofs–as–programs paradigm. In our work, we created a simple type theory that has the expressive power of Minsky's frames [5]. We define classes as *record types*. The form of a record type is $\{l_1 : T_1, \ldots, l_n : T_n\}$ where each l_i is a label and each T_i is a type. We can think of labels as slots and the type of each label as a facet. There is nothing novel about this aspect of our type theory. The novelty of our type theory is our definition of *compatible subtypes*. The compatible subtypes relation is extension of the well–known structural subtyping relation. Under structural subtyping, a class C is a subtype of C', denoted $C \prec C'$, if all the fields of C' are fields of C. More formally, if $C = \{l_1 : T_1, \ldots, l_n : T_n\}$ and $C' = \{l_1 : T_1, \ldots, l_m : T_m\}$ for $m \leq n$, then $C \prec C'$. We extend this relation to allow labels to be synonyms of each other. We use a controlled vocabulary to determine synonyms. The controlled vocabulary is specified by parameterizing the fields of a class with ranges of labels. In other words, a class can have a variable for a label that is bound to a range of labels. For example, the class in (1) can use either the labels hue or color for primary colors.

$$\Lambda x : [\mathsf{hue}, \mathsf{color}].\{x : \mathsf{Primary\ Colors}, \mathsf{weight} : \mathsf{Pounds}\} \quad (1)$$

The class in (1) says that the slots hue and color can be interchanged when they have Primary Colors as a facet. We say that any two classes created by instantiating the label variables of a parameterize class are compatible. We extend the subtyping relation to include the notion of type compatibility. In particular, C is a *compatible subtype* of D if C is compatible to C' and D is compatible to D' and C' is a subtype[1] of D'. This means that there is a controlled vocabulary in which we can determine that all of the slots in D are equal to or synonyms of all or some of the slots in C.

In our type theory, we define inference rules for proving whether a type is a compatible subtype of another type. We also define expressions for converting objects from one type to another type. The inference rules are defined in such a way that if one can prove that C is a compatible subtype

[1]Our subtyping relation combines structural subtyping and *behavioral subtyping* [4]. Under behavioral subtyping, A can be a subtype of B and not have any fields in common. A is considered a subtype of B because in all contexts members of A *behave* the same as the members of B. For example, a Java Vector is a behavioral subtype of an array.

WWW2005, May 10–14, 2005, Chiba, Japan.
.

of D, then there is an expression that can convert objects between C and D in a meaningful way.

Our type theory provides a simple illustration of how type theory could be used in the Semantic Web services. Our type theory shows how the proofs–as–programs paradigm can be used in SWS to obtain the *proofs–as–converters* paradigm. In the proofs–as–converters paradigm, a creator of a Web service S will define the conceptually meaning of the inputs and outputs as types. If another Web service T wanted to give an object to S as input, it would need to construct a proof that T and S are compatible. A converter would be extracted from the proof that converts the objects between the services. By creating decidable type theories, it will be possible to have machines automatically resolve heterogeneity in real–time. But what is the likelihood of having decidable type theories that are expressive enough? Will we be able to do the same kinds of conceptual modeling using type theory that is possible in Description Logics? I hope during the Semantics workshop, we can address these questions and begin considering type theory as a foundation for reasoning about services.

2. REFERENCES

[1] J. Bates and R. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7:113–136, 1985.

[2] R. L. Constable. Constructive mathematics and automatic program writers. In *IFIP Congress*, pages 229–233. North-Holland, 1971.

[3] W. Howard. The formulae-as-types notion of construction. In J. Seldin and J. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[4] B. H. Liskov and J. M. Wing. Family values: A semantic notion of subtyping. *ACM Transactions on Programming Languages and Systems*, pages 1811–1841, Nov. 1994.

[5] M. Minsky. A framework for representing knowledge. In P. Winston, editor, *The Psychology of Computer Vision*. McGraw Hill, New York, 1975.

[6] R. Moten. Toward a type–theoretic approach for resolving data heterogeneity in multiple attribute negotiations. Technical Report 02-4, Rensselaer Polytechnic Institute, 2002. Online at http://cs.colgate.edu/faculty/rod/www.cs.rpi.edu/ motenr/rpi.ps.

[7] R. Moten. Using type theory as a language for negotiation objects in online exchanges. Technical Report 02-10, Rensselaer Polytechnic Institute, 2002. Online at http://www.cs.rpi.edu/tr/02-10.pdf.

[8] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquim 73*, pages 73–118. Studies in Logic and the Foundations of Mathematics, Vol. 80, North-Holland, 1973.

[9] Per Martin-Löf. Constructive mathemetics and computer programming. In *Logic, Methodolgy, and Philosophy of Science VI*, pages 153–175. North-Holland, 1982.