A short article for the

Encyclopedia of Artificial Intelligence: Second Edition

"Logic, Higher-order"

by Dale Miller, February 1991

While first-order logic has syntactic categories for individuals, functions, and predicates, only quantification over individuals is permitted. Many concepts when translated into logic are, however, naturally expressed using quantifiers over functions and predicates. Leibniz's principle of equality, for example, states that two objects are to be taken as equal if they share the same properties; that is, $a = b$ can be defined as $\forall P[P(a) \equiv P(b)]$. Of course, first-order logic is very strong and it is possible to encode such a statement into it. For example, let *app* be a first-order predicate symbol of arity two that is used to stand for the application of a predicate to an individual. Semantically, $app(P, x)$ would mean P satisfies x or that the extension of the predicate P contains x. In this case, the quantified expression could be rewritten as the first-order expression $\forall P[app(P, a) \equiv app(P, b)]$ (appropriate axioms for describing *app* are required). Such an encoding is often done in a multi-sorted logic setting, where one sort is for individuals and another sort is for predicates over individuals. Set-theory is another first-order language that encodes such higher-order concepts using membership $\in$ as the converse of *app*. Higher-order logics arise from not doing this kind of encoding: instead, more immediate and natural representation of higher-order quantification are considered. Indeed naturalness of higher-order quantification is part of the reason why higher-order logics were initially considered by Frege and Russell as a foundation for mathematics.

SYNTAX OF HIGHER-ORDER LOGIC

A common approach to describing the syntax of a higher-order logic is to introduce some kind of typing scheme. One approach types first-order individuals with ι, sets of individuals with $\langle\iota\rangle$, sets of pairs of individuals with $\langle\iota\iota\rangle$, sets of sets of individuals with $\langle\langle\iota\rangle\rangle$, etc. Such a typing scheme does not provide types for function symbols. Since in some treatments of higher-order logic, functions can be represented by their graphs, *i.e.* certain kinds of sets of ordered pairs, this lack is not a serious restriction. Identifying functions up to their graphs does, of course, treat functions extensionally, something that might be too strong in some applications. (A logic is extensional if whenever two predicates or two functions are equal on all their arguments, they themselves are equal.) A more general approach to typing is that used in the Simple Theory of Types (Church, 1940). Here again, the type ι is used to denote the set of first-order individuals, and the type o is used to denote the sort of booleans, $\{true, false\}$. In addition to these two types, it is possible to construct *functional types*: if σ and τ are types, then $\sigma \to \tau$ is the type of functions from objects of type σ to objects of type τ. Thus, an expression of type $\iota \to \iota$ represents a function from individuals to individuals. Similarly, an expression of type $\iota \to o$ represents a function from

individuals to the booleans. Using characteristic functions to represent predicates, this latter type is used as the type of predicates whose one argument is an individual. Similarly, an expression that is of the type o is defined to be a formula. Typed expressions are built by application (if M is of type $\sigma \to \tau$ and N is of type σ, then their application $(M\ N)$ is of type τ) and abstraction (if x is a variable of type σ and M is of type τ, then the abstraction $\lambda x\ M$ is of type $\sigma \to \tau$). Equality on such λ-terms is taken to include at least the conversions rules of alphabetic change in bound variables (α-conversion) and the substitution of a λ-bound variable with an argument to which it is applied (β-conversion, namely, relating $(\lambda x\ M)N$ to $M[x/N]$).

Propositional connectives can be added to these terms by introducing the constants $\wedge$, $\vee$, and $\supset$ of type $o \to o \to o$ and $\neg$ of type $o \to o$. Expressions such as $((\wedge M)N)$ are generally written in the more usual infix notation $M \wedge N$. Quantification arises by adding (for each type σ) constants $\forall_\sigma$ and $\exists_\sigma$ both of type $(\sigma \to o) \to o$. The intended denotation of $\forall_\sigma$ is the set that contains one element, namely the set of all terms of type σ. Thus, the expression $\forall_\sigma \lambda x\ M$ (where x is a variable of type σ and M is a formula) is intended to be true if the $\lambda x\ M$ denotes the set of all members of type σ; that is, if for all x of type σ, M is true. For this reason, the above expression is abbreviated as $\forall_\sigma x\ M$. Similarly, setting the intended meaning of $\exists_\sigma$ to be the collection of all non-empty subsets of type σ yields the existential quantifier. This approach to formulating logic elegantly integrates formulas and terms into a smooth, functional framework.

SEMANTICS OF HIGHER-ORDER LOGIC

There are many ways to interpret higher-order logic and category theory provides one of the richest possibilities (Lambek & Scott, 1986). Here we outline an early approach used by Henkin (1950). Higher-order logic can be interpreted over a pair $\langle \{\mathcal{D}_\sigma\}_\sigma, \mathcal{J} \rangle$, where σ ranges over all types. The set $\mathcal{D}_\sigma$ is the collection of all semantic values of type σ and $\mathcal{J}$ maps (logical and non-logical) constants to particular objects in their respectively typed domain. Thus, $\mathcal{D}_\iota$ is the set of all first-order individuals, $\mathcal{D}_o$ is the set $\{\mathit{true}, \mathit{false}\}$, $\mathcal{D}_{\iota \to o}$ is the set of characteristic functions of subsets of $\mathcal{D}_\iota$, etc. The mapping $\mathcal{J}$ must send the logical constants to their intended meanings; for example, $\mathcal{J}(\wedge)$ is the curried function that returns *true* when its arguments are both *true*, and *false* otherwise. A *standard model* is one in which the set $\mathcal{D}_{\sigma \to \tau}$ is the set of *all* functions from $\mathcal{D}_\sigma$ to $\mathcal{D}_\tau$. Such models are completely determined by supplying only $\mathcal{D}_\iota$ and $\mathcal{J}$. If $\mathcal{D}_\iota$ is denumerably infinite, then $\mathcal{D}_{\iota \to o}$ is uncountable: standard models can be very large. In fact, if $\mathcal{D}_\iota$ is infinite, it is possible to build a model Peano's axioms for the non-negative integers. As a corollary of Gödel's incompleteness theorem, the set of true formulas in such a standard model are not recursively axiomatizable; that is, there is no theorem proving procedure that could (even theoretically) uncover all true formulas. This interpretation of higher-order logic as denoting truth in a standard model is often used by those studying the mathematical properties of integers and structures that can be built from them (Shapiro, 1985).

It is possible to interpret higher-order formulas over domains other than the

standard one. Henkin (1950) developed a notion of *general model* that included non-standard as well as standard models. In the general setting, it is possible for $\mathcal{D}_{\sigma\to\tau}$ to be a proper subset of the set of all functions from $\mathcal{D}_\sigma$ to $\mathcal{D}_\tau$ as long as there are enough functions to properly interpret all expressions of the language of type $\sigma \to \tau$. Henkin's completeness result is then: a higher-order formula is valid in all general models if and only if it has a proof (possibly involving the axiom of extensionality). Thus, considered from the point of view of general models, higher-order logic with extensionality can be given a completeness result and this avoids the negative result due to Gödel's incompleteness results. The cost of this completeness result is giving up the desire to model only the standard model. Since that model is uncountable and includes functions and predicates that are not computable, such a cost is acceptable in many areas of computer science and artificial intelligence.

PROOF THEORY OF HIGHER-ORDER LOGIC

In (Church, 1940), a series of axioms were presented to describe the higher-order logic called the Simple Theory of Types. The first six axioms (which do not include extensionality) describe a logic that extends first-order logic by permitting quantification at all types and by replacing first-order terms by simply typed λ-terms modulo β and η-conversion (if M is of functional type then it is η-convertible to $\lambda x(Mx)$, provided x is not free in M). Many standard proof-theoretic results – such as cut-elimination (Girard, 1986), unification (Huet, 1975), resolution (Andrews, 1971), and Skolemization and Herbrand's Theorem (Miller, 1987) – have been formulated for this fragment. Using these results as a foundation, it is possible to write theorem provers for this fragment of higher-order logic (Andrews, *et.al.*, 1984). The presence of predicate quantification, however, can make theorem proving very challenging. In first-order logic, the result of substituting into an expression does not change its logical structure. In the higher-order setting, however, universal instantiation may increase the number of logical connectives and quantifiers in a formulas. For example, if P in the expression $[\ldots \wedge (Pc) \wedge \ldots]$ is substituted with $\lambda x[\exists w(Axw \supset Bww)]$ then the resulting expression (after doing β-conversion) would be $[\ldots \wedge [\exists w(Acw \supset Bww)] \wedge \ldots]$, which has one new occurrence each of a quantifier and logical connective. Theorem provers in first-order logic need to only consider substitutions that are generated by the unification of atomic formulas. Since logical connectives within substitutions are possible in higher-order logic, as this example shows, atomic formula unification does not suggest enough substitution terms.

COMPUTATIONAL APPLICATIONS OF HIGHER-ORDER LOGIC

Computational systems based on higher-order logic have recently been built and applied to subjects such as natural language parsing and theorem proving. In many of these cases, a treatment based on higher-order logic can provide very flexible and perspicuous implementations. For example, a theoretical understanding of meaning in natural language is frequently based on higher-order logic: Montague's compositional semantics for natural language is a good ex-

ample (Dowty, Wall, & Peters, 1981). Forcing the implementer to encode the theoretician's meanings into first-order logic can place a great distance between theory and implementation and can detract from the clarity of such implementations. Using a higher-order version of logic programming (Nadathur & Miller, 1990), for example, can remove some of the need for these encodings. Computational higher-order logics have also been used as a kind of meta-language for the implementation of theorem provers. In higher-order logics with λ-abstractions within terms (such as the Simple Theory of Types), bound variables are handled by the logic via α, β, and η-conversion (in comparison to, say, a first-order system such as Prolog where bound variables are handled only by programmer supplied clauses). From the point-of-view of implementing an object logic using such λ-terms, substitutions and details concerning bound variable names and scopes are all handled by the meta-logic's notions of conversion. As a result, the specification of a wide range of theorem provers can be achieved rather elegantly. A couple computer systems (Nadathur & Miller, 1988; Paulson, 1990) have been developed using fragments of the Simple Theory of Types as their foundation and used to specify and implement theorem provers in this meta-level fashion. The logic underlying these computer systems is restricted enough that unification of atomic formulas does, in fact, suggest all substitutions that need to be considered in making deductions.

The textbook (Andrews, 1986) and the handbook article (van Benthem & Doets, 1983) are good sources for getting more information on higher-order logic. Higher-order logic and intuitionistic type theory overlaps in many ways. The textbook (Nordstrom, Petersson, & Smith, 1990) is a good source for intuitionistic type theory.

BIBLIOGRAPHY

P. Andrews, D. Miller, E. Cohen, F. Pfenning, Automating Higher-Order Logic in *Automated Theorem Proving: After 25 Years*, AMS Contemporary Mathematics Series **29** (1984).

P. Andrews, *Resolution in Type Theory*, Journal of Symbolic Logic **36** (1971), 414 – 432.

P. Andrews, *An Introduction to Mathematical Logic and Type Theory*, Academic Press, 1986.

J. van Benthem and K. Doets, Higher-order logic, in *Handbook of Logic for Computer Science I*, edited by D. Gabbay and F. Guenthner, Reidel (1983), 275 – 329.

A. Church, A Formulation of the Simple Theory of Types, Journal of Symbolic Logic **5** (1940), 56 – 68.

D. Dowty, R. Wall, and S. Peters, *Introduction to Montague Semantics*, D. Reidel Publishing Co., Boston (1981).

J.-Y. Girard, The System F of Variable Types: Fifteen Years Later, Theoretical Computer Science **45** (1986), 159 – 192.

L. Henkin, Completeness in the theory of types, Journal of Symbolic Logic **15** (1950), 81 – 91.

G. Huet, A Unification Algorithm for Typed λ-Calculus, Theoretical Computer Science **1** (1975), 27 – 57.

J. Lambek and P. J. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge University Press, 1986.

D. Miller, A Compact Representation of Proofs, Studia Logica **46** (1987), 347 – 370.

G. Nadathur and D. Miller, An Overview of λProlog, eds. K. A. Bowen and R. A. Kowalski, Fifth International Logic Programming Conference, Seattle, Washington, MIT Press, August 1988, 810 – 827.

G. Nadathur and D. Miller, Higher-order Horn clauses, Journal of the ACM **37** (1990), 777 – 814.

B. Nordstrom, K. Petersson, and J. M. Smith, *Programming in Martin-Löf's type theory: an introduction*, Oxford: Clarendon, 1990.

L. Paulson, Isabelle: The Next 700 Theorem Provers, in *Logic and Computer Science*, ed. P. Odifreddi, Academic Press, 1990, 361 – 386.

S. Shapiro, Second-order languages and mathematical practice, Journal of Symbolic Logic **50** (1985), 714 – 742.