Towards a Categorical Semantics of Type Classes

Barney P. Hilken and David E. Rydeheard *

Abstract. This is an exercise in the description of programming languages as indexed categories. Type classes have been introduced into functional programming languages to provide a uniform framework for 'overloading'. We establish a correspondence between type classes and comprehension schemata in categories. Coherence results allow us to describe subclasses and implicit conversions between types.

In programming, there is a temptation to classify types by their attributes. Some versions of ML distinguish equality types (types with a given equality on their values) from other types. Such classifications prevent the proliferation of arguments which should properly be derived from types. The mechanism for this is 'overloading', in which a variety of functions is given a single name and types resolve the ambiguity. A systematic treatment which allows the definition of type classes and function overloading has become available in languages such as Haskell [Hudak et. al. 88]. Type classes should be distinguished from program 'modules' as present in, for example, ML.

The aim of this paper is to give a category-theoretic account of languages with type classes. We start with the observation (see, for instance, [Pitts 87], [Seely 87] and [Hyland, Pitts 89]) that languages with terms, types and kinds correspond to indexed categories. Kinds are objects in the base category, types are objects in the fibres and terms are arrows in the fibres. Arrows in the base are built from type constructors. The relationship between types and type classes we describe as a comprehension schema in the indexed category. Comprehension schemata, introduced by Lawvere [1970], capture the universality of set comprehension, whereby subsets are defined through predicates as \( \{ x : A \mid \phi \} \) if \( x \in A \land \phi(x) \). In type theory, type classes are \( \Sigma \)-types of the form of a kind-sum of types whose result is a kind (using Barendregt's [1989] notation the sorting is \( (\Box, \ast, \square) \)). The universality of this 'sum' defines the comprehension schema.

In addition to comprehension, an account of type classes should admit a notion of 'inclusion' or 'subsumption'. Languages with type inclusion (or subtyping) are well established. Here we elevate the inclusion to the level of kinds so as to allow subclasses. This is implicit in the following 'class elimination' rule:

\[
\Gamma; \Theta \vdash S : \{ a : K | o_1 : T_1, \ldots, o_n : T_n \} \implies \Gamma; \Theta \vdash S : K
\]

Such rules separate form and formation so that the structure of a sequent no longer determines its derivation. Coherence is the means by which we restore the link between sequents and their derivations [Mac Lane 82]. In this paper, we consider an equational theory of indexed

*This is an extended abstract. A full paper is available from the authors at the address: Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K.
categories with comprehension schemata and present a decision procedure based on conversion to canonical forms. An interesting aspect of the development is that the language of type classes we present corresponds to the canonical forms of this equational theory. The correspondence arises through type inference in that a language with subsumption is converted into a language with explicit conversions between types. We define a semantics of type classes, using the coherence to ensure that the semantics of derivable sequents is defined and independent of their derivation.

The language defined in this paper admits type classes, including classes of multiple types (n-ary type classes), and an implicit conversion between types, from subclasses to superclasses. The language is extensible in the sense that it is defined in terms of a family of function constants and type constructors. The constants may take type arguments so that there is a form of explicit (parametric) polymorphism. The implicit polymorphism of languages like ML and Haskell introduces further coherence requirements which interact with those of type classes. The requirement for coherence means that modularity becomes compromised: subtypes and subkinds cannot be treated simply as additional features of a language and incorporated in a strictly modular fashion. The approach here admits extensibility; for instance we could include function types via cartesian closure and this would not interfere with coherence. However, additional features which themselves incorporate subsumption lead us, in general, into the difficult problem of combining coherence results. The Haskell group have addressed this problem with syntactic characterisations of well-formed contexts to prevent ambiguity arising. A further restriction in the language is that variables occur at most once in contexts, so that natural contexts such as \( a : \text{Eq}, a : \text{Print} \) (declaring a type \( a \) to support an equality function and a print function) are not allowed. This somewhat simplifies the treatment of the language but, we believe, is not essential to our treatment.

Categorical semantics enables us to isolate several layers of structure in languages (see for example [Jacobs, Moggi, Streicher 91] and [Moggi 91]). The category of basic structures (e.g., categories, indexed categories, fibrations etc) required to model the language captures the context structure and general interdependency of the levels of the language (called the ‘setting’ by Jacobs). Within the language, constructs such as type and term constructors (‘features’) determine a monad on this category. Finally, a specific language is an algebra of this monad. As an illustration, consider finite products. These have two roles in modelling languages; firstly, as part of the setting, they model contexts of more than one variable; secondly, where present, they model product types (where they are part of the features and therefore described as a monad).

If we consider evaluation as part of the structure of a language then, because we evaluate terms but not contexts, these two products are different; for instance those modelling product types may be lax. A similar distinction can be drawn between two roles for comprehension schemata in describing languages. Ehrhard [1988] and Jacobs [1990], use comprehension schemata (as ‘settings’ to capture the structure of languages with dependent types. Here, we consider comprehension schemata as modelling type classes (a ‘feature’). Other work related to that of this paper is that of Curien and Ghelli [Curien 90], [Curien, Ghelli 90], especially that on the coherence of subsumption. Nipkow and Snelting use order-sorted unification for type inference in languages with type classes [Nipkow, Snelting 90].

**Comprehension**

For indexed category \( p : C^\to \to \text{Cat}, C \) is the base category and \( p(K) \) the fibre over \( K \). For arrow \( s \) in the base, denote the functor \( p(s) \) by \( p_s \), or, where there is only one indexed category under consideration, by \( s^* \). Composition of arrows in categories is written as ‘;’ in diagrammatic