Representing the Abstraction Theorem in Agda

Robert J. Simmons, Daniel R. Licata

In the second section of John Reynolds’ 1983 paper Types, Abstraction and Parametric Polymorphism, he describes a typed programming language with higher-order functions, pairs, and boolean variables and gives a denotational semantics to the well-typed terms. A key feature is that free type variables are allowed, and the denotational semantics requires that abstract types be mapped to sets.

In the third section of that paper, he gives a relational semantics that describes a relationship between terms with different (but related) mappings of abstract types to sets, and states the Abstraction Theorem: a expression can always be related to itself, even under two different (but related) mappings on abstract types to sets. This theorem is the foundation of parametricity in a functional programming language.

The surprising and beautiful outcome of formalizing this result in Agda, a Haskell-like dependently typed functional programming language, is that these two sections in Reynolds are almost perfectly parallel to one another, and the proof of the Abstraction Theorem runs parallel to the definition of the language’s semantics.

Building expressiveness into the metalanguage

In order to describe the semantics of a language with functions, pairs, units, and boolean variables, we must have a language that describes what the set of unit values, Boolean values, pair values, and function values are. Functions are already built in to Agda, we define the rest.

The "meaning" interpretation interprets types as Agda types (which we can think of as sets of Agda sequent). When we interpret a derivation, the result is a map from the interpretation of the premises to the interpretation of the conclusion.

The "relational" interpretation interprets types as relations between Agda types, so for every variable in a type context we need to provide two Agda types to give that context a type.

The "interpretation" interprets types as functions from type variables assignments to maps of variables to Agda types.

Finally, the meaning of an expression is a function from the meaning of its type context (a mapping of type variables to Agda types) and the meaning of its variable context (a mapping of expression variables to Agda values of the appropriate type) to an Agda value of whatever Agda type the expressions type translates to under the interpretation of the type context.

The "relational" interpretation interprets types as relations between Agda types, so for every variable in a type context we need to have two Agda types and a relation between them. Types are functions from the interpretation of type contexts (two functions S1 and S2 mapping type variables to Agda types and one function in mapping type variables to relations between the two types designated by the other functions) to two Agda types (specifically, the types given by the meaning function on types) and a relation between them.

There is a very regular pattern to both the relationship between the two interpretations and the way in which the relational interpretation relies upon the meaning interpretation.

We usually read the Abstraction Theorem as saying that typed expressions are always related to themselves; this reading says that an expression is interpreted as a function from the meaning of the type context (mapping type variables to relationships between types) and the meaning of the variable context (mapping expression variables to relations between Agda values) to a relation between the meaning of the expression given by eval in the two related environments.