Polymorphic variants in dependent type theory

Claudio Sacerdoti Coen[1]* and Dominic Mulligan[1]

Dipartimento di Informatica – Scienza e Ingegneria,
Università di Bologna,
Mura Anteo Zamboni 7, Bologna (BO) Italy

The expression problem and polymorphic variants The expression problem is maybe the best known issue in programming language development. The problem consists in the extension of a data type of expressions to include new constructors. All operations defined on the data type have to be updated to cover also the new constructors. The difficulty is that the extension should be done modularly: it should be possible to add new constructors or to merge together two sets of them without modifying the already developed code. Object oriented languages provide a solution to the problem: the type of expressions becomes an interface that is implemented by each constructor, represented as a class. The tradeoff is the impossibility to close the universe of expressions in order to perform pattern matching and exhaustivity checking. Pattern matching, in particular, seems the natural way to reason on (expression) trees.

A partial solution can be obtained using functional languages based on algebraic types and pattern matching, like OCaml or Haskell. The idea consists in "opening" the algebraic data type E of expressions by turning it into a parametric type $E\ \alpha$ where the type parameter α replaces E in the recursive arguments of the constructors of E. The recursive type $\mu\alpha.(E\ \alpha)$ is then isomorphic to the original "closed" type E. In order to merge together two types of expressions $E_1\ \alpha$ and $E_2\ \alpha$ is it sufficient to build the parametric disjoint union $E\ \alpha := K_1 : E_1\ \alpha \mid K_2 : E_2\ \alpha$. Similarly, given two functions f_1, f_2 typed as $f_i : (\alpha \to \beta) \to E_i\ \alpha \to E_i\ \beta)$, it is possible to build the function f over $E\ \alpha$ by pattern matching over the argument and dispatching the computation to the appropriate f_i.

The previous solution has several major problems. The first one is the non associativity of the binary merge operation, which is a major hindrance to modularity. Concretely, it is often the case that one needs to explicitly provide functions to convert back and forth between isomorphic types. A second problem is the following: merge is implemented on top of a *disjoint* union, when the expected operation is the standard union. Again, this is a problem for modularity, since it disallows multiple inheritance. Finally, the solution is not efficient since every merge operation adds an indirection that is paid for both in space (memory consumption) and pattern matching time.

A satisfactory solution to the expression problem for functional languages is given by *polymorphic variants*, like the ones implemented by Guarrigue in the OCaml language [**?**, **?**] . The idea is to add to the programming language a (partial) merge operation over algebraic types that corresponds to a standard union. Merging fails when the the same constructor occurs in both types to be merged with incompatible types. Replaying the previous construction with this operation already gives a satisfactory solution by solving at once all previous problems. Moreover, polymorphic variants and their duals, polymorphic records, allow for an interesting typing discipline where polymorphisms is obtained not by subtyping, but by uniformity. For

*The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881.

example, a function could be typed as $[K_1 : T_1 \mid K_2 : T_2] \cup \rho \to T$ to mean that the input can be any type obtained by merging a type ρ into the type of the two listed constructors. The function can be applied to a $K_3\ M$ by instantiating (via unification in ML) ρ with $[K_3 : T_3] \cup \sigma$ for some σ and for $M : T_3$.

An efficient encoding in dependent type theory In the talk we will show an *efficient* encoding of *bounded polymorphic variants* in a dependent type theory augmented with implicit coercions. The languages of Coq and Matita, for instance, can be used for the encoding, doing everything at user level. We name bounded polymorphic variants the class of all polymorphic variants whose constructors are a subset of one (or more) sets of typed constructors — called universes — given in advance.

Several encodings are possible. However, we will limit ourselves to the one that respects the following requirements:

1. Universe extension: after adding a new constructor to a universe, all files that used polymorphic variants that were subsets of the universe should typecheck without modifications. Therefore, the restriction to the bounded case is not problematic because the merge of two universes does not require any changes to the code.

2. Efficiency of extracted code: after code extraction, bounded polymorphic variants and classical algebraic types should be represented in the same way and have the same efficiency.

3. Expressivity: all typing rules for polymorphic variants discussed in the literature must be derivable. In particular, each bounded polymorphic variant should come with its own elimination principle that allows to reason only on the constructors of the universe that occur in the polymorphic variant.

4. Non intrusivity: thanks to implicit coercions and derived typing rules, writing code that uses polymorphic variants should not require more code than what is required in OCaml.

Our encoding is based on the following idea: a universe is represented as a standard algebraic data type; a polymorphic variant on that universe is represented as a pair of lists of constructors, those that may and those that must be present; dependent types and computable type formers allow to turn the two lists into the sigma-type of inhabitants of the universe that are built only from constructors that respect the constraints; code extraction turns the sigma-type into the universe type, ensuring efficiency; implicit coercions are used to hide the sigma type construction, so that the user only works with the two lists; more dependently typed type formers compute the type of the introduction and elimination rules for the polymorphic variants; the latter are inhabited by dependently typed functions. All previous functions and type formers cannot be expressed in the type theory itself. However, we provide a uniform construction (at the meta-level) to write them for each universe.

Finally, we will show how the same ideas can be exploited for a similar efficient representation of polymorphic records in dependent type theory.