Technical Writing -- Structure Investigation

Transition Words of “Relaxing the Value Restriction (Jacques Garrigue, 2004)”

**(Writing logic: logic division of ideas)** Restricting poly-morphism to values, as Wright suggested [19], is now the standard way to obtain soundness in ML-like programming languages with imperative features. Section 2 explains how this conclusion was reached. This solution’s main advantages are its utter simplicity (only the generalization rule is changed from the original Hindley-Milner type system), and the fact it avoids distinguishing between applicative and imperative type variables, giving identical signatures to pure and imperative functions.

**(Writing logic: contrast)** Of course, this solution is sometimes more restrictive than previous ones: by assuming that all functions may be imperative, lots of polymorphism is lost. However, this extra polymorphism appeared to be of limited practical use, and experiments have shown that the changes needed to adapt ML programs type checked using stronger type systems to the value only polymorphism type system were negligible.

**(Writing logic: none)**

Almost ten years after the feat, it might be useful to check whether this is still true. Programs written ten years ago were not handicapped by the value restriction, but what about programs we write now, or programs we will write in the future?

In his paper, Wright considers 3 cases of let-bindings where the value restriction causes a loss of polymorphism.

**(Writing logic: logical division of ideas)**

1. Expressions that never return. They do not appear to be really a problem, but he remarks that in the specific case of ∀α.α, it would be sound to keep the stronger type.

2. Expressions that compute polymorphic procedures.This amounts to a partial application. Analysis of existing code showed that their evaluation was almost always purely applicative, and as a result one could recover the polymorphism through eta-expansion of the whole expression, except when the returned procedure is itself embedded in a data structure.

3. Expressions that return polymorphic data structures. A typical example is an expression returning always the empty list. It should be given the polymorphic type α list, but this is not possible under the value restriction if the expression has to be evaluated.

**(Writing logic: contrast)** Of these 3 cases, the last one, together with the data structure case of the second one, are most problematic: there is no workaround to recover the lost polymorphism, short of recomputing the data structure at each use. This seemed to be a minor problem, because existing code made little use of this kind of polymorphism inside a data structure. However we can think of a number of cases where this polymorphism is expected, sometimes as a consequence of extensions to the type system.

**(Writing logic: logical division of ideas)**

1. **(Writing logic: logical division of ideas)** Constructor and accessor functions. While algebraic datatype constructors and pattern matching are handled specially by the type system, and can be given a polymorphic type, as soon as we define functions for construction or access, the polymorphism is lost. The consequence is particularly bad for abstract datatypes and objects [15], as one can only construct them through functions, meaning that they can never hold polymorphic values.

2. **(Writing logic: logical division of ideas)** Polymorphic variants [2]. By nature, a polymorphic variant is a polymorphic data structure, which can be seen as a member of many different variant types. If it is returned by a function, or contains a computation in its argument, it looses this polymorphism.

3. **(Writing logic: logical division of ideas)** Semi-explicit polymorphism [4]. This mechanism allows to keep principality of type-checking in the presence of first-class polymorphism. This is done through adding type variable markers to first-class polymorphic types, and checking their polymorphism. Unfortunately, value restriction looses this polymorphism. A workaround did exist, but the resulting type system was only “weakly” principal.

**(Writing logic: none)** We will review these cases, and show how the value restriction can be relaxed a little, just enough for many of these problems to be leveled. As a result, we propose a new type system for ML, with relaxed value restriction, that is strictly more expressive (it types more programs) than ML with the usual value restriction.

**(Writing logic: logical derivation)**

The starting point is very similar to the original observation about ∀α.α : in some cases, polymorphic types are too generic to contain any value. As such they can only describe empty collections, and it is sound to allow their generalization.

Our basic idea is to use the structural rules of subtyping to recover this polymorphism: by subsumption, if a type appears only in covariant positions in the type of a value, it shall be safe to replace it with any of its super types. From a set-theoretic point of view, if this type is not inhabited,then it is a subtype of all other types (they all contain the empty set). If it can be replaced by any type, then we can make it a polymorphic variable. For instance, consider this expansive binding:

val f : unit -> ’\_a list

The variable ’\_a is non-generalizable: it can be instantiated only once, and is shared between all uses of f. We can replace ’\_a by the base type zero. Assuming that zero is not inhabited, it is sound to replace all its covariant occurrences by polymorphic variables:

val f : unit -> ’a list

Since ’\_a had only covariant occurrences, zero does not appear in this new type, making it strictly more general than the original one.

**(Writing logic: contrast and reasoning)**

Unfortunately, this simple reasoning cannot be translated into a direct proof : we are aware of no set theoretic model of ML extended with references. Nonetheless this intuition will lead us to a semi-syntactic proof using semantic types.

As an interesting aside to this result, we will see that the resulting system, while being sound, does not enjoy the subject reduction. This may explain why it was not considered to date.

**(Writing logic: ordered...but not chronologically)** This paper is organized as follows. After a short reminder on why the value restriction became so popular, we give some examples of our scheme applied to simple cases,and then show how it helps solving the problems described above. In section 5 we formalize our language and type system, and prove its soundness using semantic types in section 6, before concluding.