Verbose Typing

Robert Ennals
rje33@cam.ac.uk
Cambridge University
(Undergraduate)

1 Motivation

In type systems that require one to manually specify types, one is constrained by practicality into having relatively short types that can be read and written easily. This constrains the amount of information that can be encoded in a type, and forces one to restrict the expressiveness of the type system. Type inference systems such as that of Haskell [1] improve things greatly by allowing the type system to infer types, but still require types to be stated when declaring type classes.

By removing such requirements to state types, we can grant ourselves the freedom to make types as large as we like, including much more information than would otherwise be practical. One way to do this is by separating the concepts of a type identifier and a type constraint.

2 A Simple Type System

In the proposed system, every object has its own independent type, and has members. These are similar to functions in Haskell type classes [1] and to dynamically typed object members. A member is identified by a unique name. The set of members of an object can be any subset of the global set of possible members, and the object may implement any member with any type. The type is independent of the member identifier.

Making the type of the member independent allows increased flexibility, especially useful in large complex systems. For example, one might have objects that wish to export windows with different abilities. This should be allowed, as long as all these windows satisfy the constraints required by the function that is using them. Fixing the constraints of the implementation of a member prevents one from being able to make further requirements or provide less guarantees.

Types are divided into type requirements and type guarantees. "If" is implemented as a special case. Instead of requiring its arguments to have the same type, it allows its arguments to have any type, and the guarantee of its result is the intersection of the guarantees of its arguments. Likewise, applying a function to its own result does not constrain the return type to be the same as the argument type.

Example constraints are that an object must have a specified member (which may also have constraints), be of a specified base type, satisfy one of a list of constraints (for pattern matches), satisfy all of a list of constraints, or satisfy the constraints imposed by another function.

Disjoint types are handled by giving one of the members an identifier type. This is a special type, that has no purpose other than to be used as an identifier in pattern matching. All disjoint types are part of a global namespace, allowing anything to take on any set of disjoint types. Pattern matches can match several possible disjoint types and place different requirements on other things, depending on which disjoint type is found. Matching is checked entirely statically with no runtime match errors.

If one has a concept of a reactive function [2], then one can also allow disjoint types to be introduced on the fly with a non referentially transparent "new" function. This allows one to do things like typing the contents of arrays. A new disjoint type is created for the array contents when a new array is created, and is preserved when the array is mapped or joined. This allows type safe use of keys between arrays.

No attempt is made to reduce the constraints into a simplified general form or remove recursion from the constaints. Type checking proceeds by expanding function evaluations until either the type constraints repeat or an error are found. In most cases, one or the other will be found fairly quickly, however in some special cases it may run for a very long time or even not terminate. One such case is a function involving a divergent computation on Church numerals. As every Church numeral has a different type in this system, there will be no fixed point or error, and so the type check will not terminate. Such cases are, however, very rare and generally not very useful. In practice problems could be avoided by giving a type error if a recursion did not reach a fixed point within a certain (large) number of expansions.

This type system attempts to get close to the freedom of Smalltalk, while maintaining the safety of strong typing.

References

[1] Paul Hudak, Phillip Wadler et al, The Haskell report 1.4. 1997.

[2] Robert Ennals, Controlled Temporal Non-Determinism for Reasoning With A Machine Of Finite Speed, ICFP98 (Poster).