MILNER78

Milner, R.;
A Theory of Type Polymorphism in Programming;
Journal of Computer and System Sciences 17 (1978) pp. 348-375.

A formal type discipline for polymorphic procedures in a simple programming language is presented. A compile time type-checking algorithm to enforce it is also given. Although a Large portion of it is necessarily technical the approach is interesting.

One of the main aims is to ensure that programs are robust while accomplishing the flexibility of allowing polymorphic procedures. This polymorphism arises largely because there are primitive polymorphic operators in every programming language including such things as assignments, function application and list-processing operators. Type constraints on these operators, in conjunction with variable declaration and use, give the type for program phrases so the type checking of them is important.

Presentation of the discipline is done through two means. Examples are presented in a metalanguage ML for which the type checker was implemented. This shows the ability to handle useful languages within the discipline. Justification of the discipline is done through a simple applicative language called Exp. This language has types which are a hierarchy of functional types over a set of basic ones thereby producing polymorphism through function application and variable binding. Extension of the idea to handle an assignment operator is easy in the syntactic sense of the discipline but mor complicated semantically.

The main results of the paper theoretically are the Semantic Soundness Theorem and the Syntactic Soundness Theorem. Correct type assignment is referred to as well-typing. The Semantic Soundness Theorem says that a well-typed program is semantically free of type violation. When there is no type violation the need to carry types at run-time is obliterated and the compile-time type-checker presented is designed to detect well typed programs. The Syntactic Soundness Theorem states that if the well-type algorithm W presented in the paper succeeds it produces a well typing of the program.

The legal type assignment produced is conjectured to be the most general one. This means that the other legal assignments are substitution instances of it. It is important to note that this does not preclude the use of explicit typing by the programmer; this should simply correspond to a substitution instance.

The author acknowledges the use of a technique proposed for unification by Robinson. The adaption is well presented and the technique is involved in the proof that W is sound.

Although the technical content is quite heavy the ideas in this paper are practical to language design and implementation. Some of the proof details are tedious (to the extent that even the author comments on it). The article provides background in the understanding of languages using polymorphic type checking and would be valuable reading for those seeking to implement such a system.