An Optimized Type Inference Algorithm with Delayed Type Substitution

Katsuhiro Ueno (410015)

School of Department of Information Science,
Japan Advanced Institute of Science and Technology

February 9, 2006

Keywords: Type Inference, Optimized Algorithm, Functional Programming Language, ML.

Type inference is a major characteristics of modern functional programming language, such as Standard ML, Haskell, ObjectiveCaml, etc. This is a feature to infer the most general type of a program. By this feature, compiler can decide the type of a given untyped program. The polymorphic type inference algorithm defined against extended lambda expression including polymorphic let expressions which allows type variable bindings for polymorphic types is the core of this feature.

Since the idea of polymorphic type inference has proven very useful in practice, range of programming languages have adopted this feature. But on the other hand, the type inference is one of the most complex and most time-consuming step among compiler front-end phases, so it makes compiler implemenation more complex and increases processing time for compilation. Development of functional and efficient type inference algorithm is an important task for next generation of programming languages equipped with the polymorphic type inferecence feature.

However, it is already shown that typing a polymorphic functional programming language is DEXPTIME-complete. Constructing an "efficient algorithm" in the meaning of computational theory is impossible. In spite of existence of this theoritical limit, practically more efficient type inference algorithm has been needed in order to speed up compilation processes. In this thesis, I propose a new functional and practically fast type inference algorithm, as a basis of reliable and efficient implemantations of state-of-the-art functional programming languages.

Milner's type inference algorithm $\mathcal{W}$, which is adopted by many recent compilers of functional programming languages, has the following problems abround its efficiency.

- $\mathcal{W}$ heavily repeats applying type substitutions to large structures including type terms such as type environments.
- While let expressions appear very often in actual program, a whole scan of type environment is needed every time to calculate a set of bounded type variables for a polymorphic type produced by each let expression.

I solved the first problem by preserving a type substitution as an explicit environment for evaluating type environment and then delaying application of type substitution. The other problem can be solved by keeping a set of free type variables which is reachable from type environment at every turn. Additionally, in order to achieve this strategy that delays type substitutions, the unification algorithm should be refined so that it can calculate a unifier under the type substitution environment.

In this thesis, I present a set of an unification algorithm $\mathcal{DU}$ and practically efficient type inference algorithm $\mathcal{DW}$ constructed on the ideas described above. I also prove the soundness and completeness of $\mathcal{DU}$, and the type soundness of $\mathcal{DW}$. $\mathcal{DW}$ seems much more efficient than $\mathcal{W}$: For example, algorithm $\mathcal{W}$ involves $O(n^2)$ time type substitutions for typing nested lambda abstractions, but in contrast, algorithm $\mathcal{DW}$ completes the typing only with $O(n)$ time substitutions.

I also implement a type inference module based on $\mathcal{DW}$ adequated to Core Syntax of Standard ML on SML$^\sharp$ compiler in order to demonstrate its practical feasibility. Furthermore, I took some benchmarks among several type inference algorithms by using this implementation and practical programs to clear that $\mathcal{DW}$ is faster than $\mathcal{W}$ in practice. The result of the benchmark shows that $\mathcal{DW}$ is intensely faster than $\mathcal{W}$, and $\mathcal{DW}$ is as fast as the well-known ad-hoc imperatively-extended type inference algorithm.

At the last of this thesis, I present some discussions about additional extensions to make $\mathcal{DW}$ more efficient.