Polymorphism, subtyping and type inference in MLsub

Stephen Dolan Alan Mycroft
University of Cambridge

We present a type system combining subtyping and ML-style parametric polymorphism. Unlike previous work, our system supports type inference and infers compact types. We demonstrate this system in the minimal language MLsub, which types a strict superset of core ML programs.

1 Introduction

The Hindley-Milner type system of ML and its descendants is popular and practical, sporting decidable type inference and principal types. However, extending the system to handle subtyping while preserving these properties has been problematic.

Subtyping is useful to encode extensible records, polymorphic variants, and object-oriented programs, but also allows us to expose more polymorphism even in core ML programs that do not use such features by more carefully analysing data flow. Consider the `select` function, which takes three arguments: a predicate p, a value v and a default d, and returns the value if the predicate holds of it, and the default otherwise:

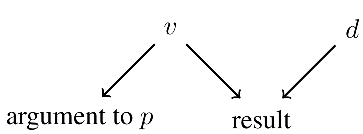
$$\texttt{select}\ p\ v\ d = \texttt{if}\ (p\ v)\ \texttt{then}\ v\ \texttt{else}\ d$$

In ML and related languages, `select` has type scheme

$$(\alpha \to \texttt{bool}) \to \alpha \to \alpha \to \alpha$$

This type is quite strange, in that it demands that whatever we pass as the default d be acceptable to the predicate p. But this constraint does not arise from the behaviour of the program: at no point does our function pass d to p.

Let's examine the actual data flow of this function:



Only by ignoring the orientation of the edges above could we conclude that d flows to the argument of p. Indeed, this is exactly what ML does: by turning data flow into equality constraints between types, information about the *direction* of data flow is ignored. Since equality is symmetric, data flow is treated as undirected.

To support subtyping is to care about the direction of data flow. With subtyping, a source of data must provide at least the guarantees that the destination requires, but is free to provide more.

In his PhD thesis, Pottier[1] noticed that the graph of data flow has a simple structure. By keeping a strict separation between inputs and outputs, we can always represent the constraint graph as a *bipartite* graph: data flows from inputs to outputs. With edges only from inputs to outputs, such graphs have no cycles (or even paths of more than one edge), simplifying analysis.

We take this insight a step further, and show that by keeping the same religious distinction between input and output we can develop a variant of unification compatible with subtyping, allowing us to infer types.

2 Input and output types

Our types form a lattice, with a least-upper-bound operator $\sqcup$ and a greatest-upper-bound operator $\sqcap$. The structure of programs does not allow the lattice operations $\sqcup$ and $\sqcap$ to appear arbitrarily. If a program chooses randomly to produce either an output of type τ_1 or one of type τ_2, the actual output type is $\tau_1 \sqcup \tau_2$. Similarly, if a program uses an input in a context where a τ_1 is required and again in a context where a τ_2 is, then the actual input type is $\tau_1 \sqcap \tau_2$. Generally, $\sqcup$ only arises when describing outputs, while $\sqcap$ only arises when describing inputs. In a similar vein, the least type $\bot$ appears only on outputs (of non-terminating programs), while the greatest type $\top$ appears only on inputs (an unused input). Thus, we distinguish *positive* types τ^+ (which describe outputs) and *negative* types τ^- (which describe inputs):

$$\begin{aligned}
\tau^+ &::= \alpha \mid \tau^+ \sqcup \tau^+ \mid \bot \mid \text{unit} \mid \tau^- \to \tau^+ \mid \mu\alpha.\tau^+ \\
\tau^- &::= \alpha \mid \tau^- \sqcap \tau^- \mid \top \mid \text{unit} \mid \tau^+ \to \tau^- \mid \mu\alpha.\tau^-
\end{aligned}$$

Positive types describe something which is *produced*, while negative types describe something which is *required*.

[1] The *mono-polarity invariant* [3, ch. 12]

3 Unification and biunification

The core operation of the Damas-Milner type inference algorithm [1] is *unification*. Unification relies on the substitution of equals for equals, which maps well to dealing with systems of equations between types. With subtyping, the standard unification algorithm does not apply, since we deal with subtyping constraints rather than type equations.

There are three different situations in which Damas-Milner inference uses unification. The first is to unify two possible output types of an expression, for instance the two branches of an `if`-expression. The second is the dual of the first, unifying two required input types of an expression when typing a λ-bound variable (all uses of which must be at the same type). With subtyping, these correspond respectively to the introduction of a $\sqcup$ or a $\sqcap$ operator. In MLsub, these cannot fail: an `if` which may produce two disparate types produces an underconstrained and useless output, an a λ-bound variable used in two disparate ways requires an overconstrained and impossible input, but neither can cause an error.

The third situation in which unification is used is the routing of inputs to outputs. For instance, the typing rule for an application $e_1 e_2$ constrains the type of the value produced by e_2 to be the same as that required by the domain of e_1. If they don't match, this *can* cause an error: passing a string to a function expecting an integer tends to end badly.

With subtyping, we demand only that the type of e_2 be a subtype of the domain of e_1. Given $e_1 : \tau_1^- \to \tau_2^+$, $e_2 : \tau_3^+$, we have the constraint $\tau_1^+ \leq \tau_3^-$. In general, our constraints are always of the form $\tau^+ \leq \tau^-$: we ensure that some value that we *produce* of type τ^+ is acceptable in some context that *requires* τ^-.

This syntactic restriction allows us to define an algorithm analagous to unification which we dub *biunification*. The difficult cases of $\tau_1 \sqcap \tau_2 \leq \tau_3$ and $\tau_1 \leq \tau_2 \sqcup \tau_3$ are excluded by construction, while the remaining cases involving lattice operations ($\tau_1 \leq \tau_2 \sqcap \tau_3$ and $\tau_1 \sqcup \tau_2 \leq \tau_3$) are easily decomposed into smaller constraints.

We then infer types using a method broadly similar to Damas-Milner inference, with biunification in place of standard unification.

4 Algebraic subtyping

Much previous work on subtyping first defines *ground types*, which are types not containing type variables, and then defines polymorphism by quantification over ground types [4, 2].

This leads to a surprisingly finicky subtyping relation between polymorphic types. Quantifying over ground types admits case analysis over types as a means of proving subtyping relationships between polymorphic types. Essentially, defining polymorphic subtyping in terms of ground types bakes in a closed-world assumption.

Instead, we reformulate subtyping by giving an algebraic axiomatisation of subtyping. Some counterintuitive subtyping relations whose truth relies on the nonexistence of certain types are thereby false in our system. Our definition uses only open-world reasoning: case analysis on a type variable is precluded.

By relating this algebraic structure to the theory of regular languages, we are able to simplify the types inferred by our system using standard algorithms from the theory of finite automata.

5 Implementation

We have implemented a simple functional language based on these ideas, which supports type inference, type simplification using automata, record types with structural subtyping and a polymorphic subsumption checker for verifying type annotations.

We used our implementation to infer and simplify types for the functions in OCaml's standard `List` module[2]. The types inferred by MLsub were as compact as those inferred by OCaml, and in most cases were syntactically identical.

The implementation is available from, and can be used interactively on the first author's website:

`http://www.cl.cam.ac.uk/~sd601/mlsub`

References

[1] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[2] J. Niehren and T. Priesnitz. Entailment of non-structural subtype constraints. In P. Thiagarajan and R. Yap, editors, *Advances in Computing Science — ASIAN'99*, volume 1742 of *Lecture Notes in Computer Science*, pages 251–265. Springer Berlin Heidelberg, 1999.

[3] F. Pottier. *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, Université Paris 7, 1998.

[4] V. Trifonov and S. Smith. Subtyping constrained types. In R. Cousot and D. A. Schmidt, editors, *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer Berlin Heidelberg, 1996.

[2]We made some changes to the `List` module to satisfy our rather primitive parser (which does not accept much of OCaml's syntactic sugar). The result remains valid OCaml, equivalent to the original.