MLF with Graphs

Didier Rémy and Boris Yakobowski - INRIA Rocquencourt

February 13, 2006

Abstract

ML^F is a type system that generalizes ML with first-class polymorphism as in System F. Building on it, we propose a new syntax for types, based on graphs. The new syntax is expected to be more intuitive for the user. It also greatly simplifies the presentation of the system; for example, equivalence on types is greatly simplified. We present a set of rules based on graphs for instantiation, which are sufficient to derive all equivalent rules on syntaxic type. We then propose a new unification algorithm, and prove that it is correct and complete.

1 Syntaxic presentation of ML^F

ML^F[LB04, LBR03] is a type system that generalizes ML with first-class polymorphism as in System F. Expressions may contain second-order type annotations. Every typable expression admits a principal type, which however depends on type annotations. Principal types capture all other types that can be obtained by implicit type instantiation and they can be inferred. All expressions of ML are well-typed without any annotations. All expressions of System F can be mechanically encoded into ML^F by dropping all type abstractions and type applications, and injecting types of lambda-abstractions into ML^F types. Moreover, only parameters of lambda-abstractions that are used polymorphically need to remain annotated. As a result, ML^F is a good compromise between the freedom offered by ML type inference and the full power of System F first-class polymorphism.

$$\begin{array}{lll} \tau ::= \alpha \mid C\,\tau_1 \,..\, \tau_n & & \text{Monotypes} \\ \sigma ::= \tau \mid \bot \mid \forall\,(\alpha > \sigma)\;\sigma \mid \forall\,(\alpha = \sigma)\;\sigma & & \text{Polytypes} \end{array}$$

Figure 1: Syntax of types

For reference, types in ML^F are presented in Fig 1. Type are viewed under contexts Q, which are used to bind type variables present in the scope. Three relations on them are defined (namely equivalence ($\equiv$), abstraction ($\sqsubseteq$) and instance ($\sqsubseteq$)). One of the difficulties in understanding ML^F stems from its rich equivalence relation, which comprises 8 rules. For examples, $\forall\,(\alpha > \bot)\;\forall\,(\beta > \forall\,(\gamma > \bot)\;\gamma \to \alpha)\;\alpha \to \beta$, the type of the function $K = \lambda x.\lambda y.x$, is equivalent to $\forall(\alpha > \bot)\forall(\eta = \forall\,(\beta > \forall\,(\gamma)\;\forall\,(\delta = \gamma \to \alpha)\;\delta)\;\alpha \to \beta)\eta$.

2 ML^{F}with graphs

2.1 Definition of graphs

In the graph presentation of ML^F, types are represented by two graphs: a structure graph, reminiscent of the way ML types with sharing are seen, and a binding graph. The graph corresponding to the type of $K = \lambda x.\lambda y.x$ is presented in Fig 2.

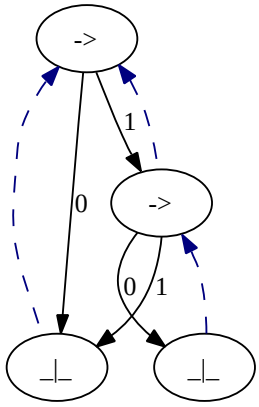

Figure 2: Type of K

More formally, a graph $G = (N, N', r, S, B)$ is composed of:

- A set of nodes N labeled by a set of constructors C or $\{\bot\}$.
- A *root* $r \in N$.
- A subset $N' \subset N$ of *bound nodes*
- A set $S \subset N \times \mathbb{N} \times N$ of *structure edges*.
- A set $B \subset N' \times N'$ of *binding edges*, labeled by $\geq$ or $=$.

For a graph to be well-formed, it must satisfy some properties; for example, arities of constructors must be respected. In particular, the binding graph must be a tree, whose leaves are labeled by $\bot$. Additional conditions must hold so that the graph can be written back into a type if needed.

2.2 Equivalence

Nearly all the equivalence relation on syntaxic types is captured by the graph representation; in particular, the graph in Fig 2 is the only one representing the type of K. Not surprisingly, the only equivalence rule not captured deals with sharing between nodes. Equivalence on graphs can be checked with a complexity of $n \cdot \alpha(n)$, where α is the inverse of the Ackermann function, and n is the size of the graphs.

2.3 Abstraction and instance

Abstraction and instance and instance, just as in the syntaxic presentation, are defined as least fixpoint of relations. Presentation with graphs is simpler: less rules are needed (2 instead of 3 for abstraction, 4 instead of 5[1]), and they are simpler to understand.

We propose an algorithm which checks whether two graphs are instance one of the other; this algorithm does not rely on the unification algorithm to work. We also show that instance derivations can be rearranged, so that rules are always used in a certain order. This greatly simplifies reasoning about derivations.

3 Unification

Given two graphs G_1 and G_2, the unification algorithm returns a graph G that is an instance of G_1 and G_2 if it exists, and fail otherwise. It can be decomposed in 3 parts:

1. **Construction of the shape of the unifier.** This is just a first-order unification on the structure graphs of G_1 and G_2.
2. **Construction of a possible binding structure.** Given the binding graphs of G_1 and G_2, and the shape of the unifier, we compute a more general binding graph for the result.
3. **Verification of instance.** We check that the operations which took place in step 1 were correct.

The algorithm is both correct (G is indeed an instance of G_1 and G_2) and complete (if a such G exists, it will be found). Moreover, since the algorithm is not recursive, termination is easy to prove, and we can easily derive a complexity bound, something which had not been previously done in ML$^{\mathrm{F}}$.

References

[LB04] Didier Le Botlan. *MLF : Une extension de ML avec polymorphisme de second ordre et instanciation implicite.* PhD thesis, Ecole Polytechnique, May 2004. 326 pages.

[LBR03] Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of System-F. In *Proceedings of the International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 27–38. ACM Press, August 2003.

[1] Rules are not presented here by lack of space.