A Constraint-Based Recast of ML–Polymorphism

(Extended Abstract)

Martin Müller* mmueller@dfki.uni-sb.de
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany

Abstract

We implement the classical Damas-Milner type inference algorithm in a calculus with higher-order abstractions and 1*st*-order constraints. We encode mono(morphic) types as constraints, and poly(morphic) types as abstractions over monotypes. Our calculus ρ_{deep} is intended as computation model of a concrete programming language, i.e., to be what the λ-calculus is for ML, or the γ-calculus [8] for Oz [2]. Unlike usual in programming language research, ρ_{deep} allows reduction below abstraction. This allows to simplify our encoding of polytypes before usage. In the envisaged programming language, the type inferencer can be run just as specified. We claim that deep reduction is one of the essential features for such a high-level implementation of type inference or abstract interpretation algorithms [4]. The algorithm can also be viewed as an extension of Wand's [9] towards let-polymorphism. Like Wand's, our algorithm is easily modified to supply other monomorphic type systems with ML style polymorphism.

The Damas-Milner Calculus

The Damas-Milner algorithm (DM, for short) for polymorphic type inference [3, 1] underlies most of today's statically typed functional programming languages. One of its most important properties is that it has principal types, i.e., there is the notion of a unique most general type for a well-typed expression. Since the DM-algorithm is by far the best-understood type inference algorithm we assume some familiarity with it.

We assume a denumerable set of variables ranged over by $x, y, z, \ldots$ or $\alpha, \beta, \gamma, \ldots$. Note that the different fonts are *only* of mnemonic importance. We consider λ-expressions M, N, monomorphic type expressions (monotypes) τ, and polymorphic ones (polytypes) σ with the abstract syntax:

$$M, N \quad ::= \quad x \mid \lambda x.M \mid M\,N \mid \mathsf{let}\ x{=}M\ \mathsf{in}\ N \qquad \tau \quad ::= \quad \alpha \mid \tau \to \tau \mid \mu\alpha.\tau \qquad \sigma \quad ::= \quad \Pi\overline{\alpha}.\tau$$

The variable x is bound in $\lambda x.M$ and $\mathsf{let}\ x{=}M\ \mathsf{in}\ N$, and α is bound in $\mu\alpha.\tau$. $\mathcal{F}(M)$ denotes the free variables of an expression. $\mu\alpha.\tau$ denotes the (possibly recursive) tree solution of $\alpha \doteq \tau$. For simplicity, we consider expressions M such that all bound variables in M are pairwise different and distinct from the free variables in M. $\tau\{\overline{\tau_0}/\overline{\alpha}\}$ denotes the capture avoiding substitution of $\overline{\tau_0}$ for $\overline{\alpha}$ into τ. An environment Γ is a finite sequence of variable-type pairs $x:\sigma$. The algorithm given by the rules below assigns types τ to expressions M wrt. an environment Γ. The following rules give the algorithm as an inference system deriving valid statements of the form $\Gamma \rhd M:\tau$. Note that this is not quite the DM-algorithm (although equivalent), since we decided to give the (LET) rule in terms of copying instead of generalisation.

$$\text{(VAR)}\quad \frac{\Gamma(x) = \Pi\overline{\alpha}.\tau}{\Gamma \rhd x : \tau\{\overline{\tau_0}/\overline{\alpha}\}} \qquad \text{(LAM)}\quad \frac{\Gamma \wedge x:\tau_1 \rhd M:\tau_2}{\Gamma \rhd \lambda x.M : \tau_1 \to \tau_2}$$

$$\text{(APP)}\quad \frac{\Gamma \rhd M:\tau_1 \to \tau_2 \qquad \Gamma \rhd N:\tau_1}{\Gamma \rhd M\,N : \tau_2} \qquad \text{(LET)}\quad \frac{\Gamma \rhd M : \tau_1 \qquad \Gamma \rhd N[M/x] : \tau_2}{\Gamma \rhd \mathsf{let}\ x{=}M\ \mathsf{in}\ N : \tau_2}$$

*Martin Müller has been supported by a fellowship from the 'Graduiertenkolleg Kognition der Universität des Saarlandes' and the Hydra Project. Appeared in: Denis Lugiez, editor. 8^{th} International Workshop on Unification. Val d'Ajol, 23–25 june, 1994. Technical Report. Université de Nancy

Type Inference in a Deep Constraint Language

Our proposal for implementing Damas-Milner polymorphic type inference is the *relational calculus* ρ_{deep} with higher-order abstractions and $1st$-order constraints. The $1st$-order constraints are used to encode monomorphic types, while polymorphic types are expressed by abstraction. The closest relative of ρ_{deep} is the ρ-calculus [7]. Unlike the ρ-calculus or its other relatives [8, 6], ρ_{deep} is not a weak calculus, i.e., it allows reduction below abstraction. This allows to simplify the abstractions encoding polymorpic types before application.

As *constraints* ϕ, ψ we pick term equations closed under composition $\wedge$ and declaration $\exists$. *Expressions* E, F are constraints, abstractions $x{:}\alpha/E$ (which can be read approximately as $\lambda\alpha.E$ with name x), or *proof obligations* $[\![M]\!]\alpha$ (which represent the constraint $\alpha \doteq \tau$ for the principal type τ of M), and closed under composition and declaration. A context is an expression with a hole $\bullet$. The $\bullet$ acts as a placeholder: $C[E]$ means context C with E replaced for $\bullet$, where capturing of free variables in E is allowed.

$$\begin{array}{lcl} \phi, \psi & ::= & \top \mid \bot \mid \phi \wedge \psi \mid \exists\alpha\, \phi \mid \alpha = \beta \mid \alpha = \beta \to \gamma \\ E, F & ::= & \phi \mid E \wedge F \mid \exists u\, E \mid x{:}\alpha/E \mid [\![M]\!]\alpha \\ C, D & ::= & \bullet \mid C \wedge E \mid \exists u\, C \mid x{:}\alpha/C \end{array}$$

Composition $\wedge$ is associative, commutative, with the neutral element $\top$. Expressions containing declarations are identified wrt. the usual scope rules for $\exists$ and α-renaming. The constraints come with a theory Δ which may in the case at hand be either the one of finite or infinite trees. This parametricity is captured by the congruence rule (EQUIV) which identifies constraints up to Δ-equivalence. The congruence rule (PROP) says that (i) constraints can be propagated downward into the bodies of abstractions, and (ii) bodies of abstractions can be simplified wrt. the context.

(EQUIV)	ϕ	$\equiv$	ψ	if $\phi \models\!\!\models_\Delta \psi$
(PROP)	$\phi \wedge x{:}\beta/E$	$\equiv$	$\phi \wedge x{:}\beta/(\phi \wedge E)$	if $\beta \notin \mathcal{F}(\phi)$

The (PROP) rule, rendering the body of an abstraction a local computation space, is an important feature of the calculus [**?**] underlying the deep guard language Oz [2]. Congruence $\equiv$ is the smallest congruence class over expressions satisfying the mentioned axioms. Reduction $\to$ is a relation on expressions modulo congruence $\equiv$. Its axioms are the rules (lam), (app), (let), and (var) below, and reduction is closed under the rule (cong).

(lam)	$[\![\lambda x.M]\!]\alpha$	$\to$	$\exists\beta\exists\gamma(\ x{:}\alpha/\alpha \doteq \beta \to \gamma \wedge [\![M]\!]\gamma\ )$
(app)	$[\![M\,N]\!]\alpha$	$\to$	$\exists\beta\exists\gamma(\ [\![M]\!]\beta \wedge [\![N]\!]\gamma \wedge \beta \doteq \gamma \to \alpha\ )$
(let)	$[\![\mathsf{let}\ x{=}M\ \mathsf{in}\ N]\!]\alpha$	$\to$	$[\![N]\!]\alpha \wedge \boxed{\exists\beta([\![x]\!]\beta)} \wedge x{:}\gamma/[\![M]\!]\gamma$
(var)	$x{:}\beta/E \wedge C[[\![x]\!]\alpha]$	$\to$	$x{:}\beta/E \wedge C[E\{\alpha/\beta\}]$
(cong)	$\dfrac{E' \equiv C[E] \quad E \to F \quad C[F] \equiv F'}{E' \to F'}$		

Rules (lam), (app), (let), and (var) essentially turn the DM-rules top-down. They encode what Wand [9] called the "action table" of its schematic algorithm. Note that we need to unfold every polytype at least once (hence $\boxed{\exists\beta([\![x]\!]\beta)}$) in order to deal with the case that for some $\mathsf{let}\ x{=}M\ \mathsf{in}\ N$, x does not appear in N. (var) is the application axiom of the ρ-calculus [7].

Polymorphic types are encoded as follows in our constraint language:

$$\begin{array}{lcll} \alpha = \sigma \to \tau & \equiv & \exists\beta\exists\gamma(\beta = \sigma \wedge \gamma = \tau \wedge \alpha \doteq \beta \to \gamma) & \alpha \parallel \{\beta, \gamma\} \\ \alpha = \mu\beta.\tau & \equiv & \exists\beta(\alpha \doteq \beta \wedge \beta = \tau) & \\ x : \Pi\overline{\alpha}.\tau & \equiv & x{:}\beta/\exists\overline{\alpha}(\beta = \tau) & \beta \parallel \{\overline{\alpha}\} \end{array}$$

The first equations are fairly clear. The encoding of polytype is a bit trickier: A polytype is encoded into an abstraction. The *universally* bound variables become *existentially* quantified variables local to its body. That is, the universal nature of a polytype is captured by the fact that an abstraction copies its body on every application with *fresh* instances of its local variables. For example, the polytype

$$x : \Pi\alpha\beta.\alpha \to \beta \to \alpha$$

is encoded as

$$x{:}\gamma/\exists\alpha\exists\beta\exists\delta(\ \gamma \doteq \alpha \to \delta \wedge \delta \doteq \beta \to \alpha\ )$$

Two applications of the same polytype abstraction (corresponding to two *different occurrences* of the *same variable*) hence will constrain its arguments differently, each time with a *different instance* of the polytype. As an example, assume two occurrences of x with respective types $\delta \doteq \mathsf{int} \to \delta'$ and $\epsilon \doteq \mathsf{bool} \to \delta' \to \epsilon'$:

$$\begin{array}{lcl} \phi & \equiv & \delta \doteq \mathsf{int} \to \delta' \wedge \epsilon \doteq \mathsf{bool} \to \delta' \to \epsilon' \wedge [\![x]\!]\delta \wedge [\![x]\!]\epsilon \quad \text{and} \\ E & = & x{:}\gamma/\exists\alpha\beta(\ \gamma \doteq \alpha \to \beta \to \alpha\ ) \end{array}$$

We obtain the following derivation:

$$\begin{array}{ll} & \phi \wedge E \\ \to^2 & \phi \wedge \exists\alpha\beta(\ \delta \doteq \alpha \to \beta \to \alpha\ ) \wedge \exists\alpha\beta(\ \epsilon \doteq \alpha \to \beta \to \alpha\ ) \wedge E \\ \equiv & \exists\beta\ (\delta \doteq \mathsf{int} \to \delta' \wedge \delta' \doteq \beta \to \mathsf{int} \wedge \epsilon \doteq \mathsf{bool} \to \delta' \to \mathsf{bool}) \wedge E \end{array}$$

The algorithm is correct in the sense that whenever τ is a principal DM-type of a closed expression M, then $[\![M]\!]\alpha \to^* C[\phi]$ such that $\phi \models\!\models \exists\overline{\beta}\alpha \doteq \tau$ where $\mathcal{F}(\tau) = \{\overline{\beta}\}$ and $\{\overline{\beta}\} \parallel \alpha$. Here, C is a context which hides intermediate type assumptions (which could also be purged by some form of garbage collection axiom).

Our correctness proof, though, is not yet complete: We can prove the correctness of the monomorphic fragment of the algorithm as well as termination. In order to cope with let-polymorphism we need the calculus to be locally confluent. It seems that there is a suitable confluent relative of the ρ- and ρ_{deep}-calculi [5]. Then, it is possible to pick a particular derivation which reduces the polymorphic to the monomorphic case (cf. rule (LET)).

A Complete Example

In this Section we give a complete derivation in order to illuminate the behaviour of our algorithm. To keep things simple, we drop type assumptions when they are no longer used. We also leave the "dummy application" of a polytype out (see rule (let)) since our example is well-behaved. We reduce the expression:

$$M \quad = \quad \lambda y.\lambda z.\lambda f.\mathsf{let}\ x{=}(\lambda u.fy)\ (fz)\ \mathsf{in}\ x$$

according to a depth first reduction stratgy: I.e., bodies of abstractions are simplified as much as possible before application. Relying on the confluence of the calculus this is correct, and it avoids to duplicate work by early application.

The first derivation few steps descend into the λ-expression accumulating type assumptions and constraints until they reach the let-expression (for brevity, we write $\exists\overline{\alpha}$ instead of $\exists\alpha_1 \ldots \exists\alpha_n$):

$$\begin{array}{ll} & [\![M]\!]\alpha \\ \to & \exists\beta\gamma \qquad (\alpha{\doteq}\beta{\to}\gamma \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ y{:}\beta{\wedge}[\![\lambda z.\lambda f.\mathsf{let}\ x{=}(\lambda u.fy)\ (fz)\ \mathsf{in}\ x]\!]\gamma) \\ \to & \exists\beta\gamma\beta'\gamma' \qquad (\alpha{\doteq}\beta{\to}\gamma{\wedge}\gamma{\doteq}\beta'{\to}\gamma' \qquad\qquad\qquad\qquad \wedge\ y{:}\beta{\wedge}z{:}\beta'{\wedge}[\![\lambda f.\mathsf{let}\ x{=}(\lambda u.fy)\ (fz)\ \mathsf{in}\ x]\!]\gamma') \\ \to & \exists\beta\gamma\beta'\gamma'\beta''\gamma'' \quad (\alpha{\doteq}\beta{\to}\gamma{\wedge}\gamma{\doteq}\beta'{\to}\gamma'{\wedge}\gamma'{\doteq}\beta''{\to}\gamma'' \wedge\ y{:}\beta{\wedge}z{:}\beta'{\wedge}f{:}\beta''{\wedge}[\![\mathsf{let}\ x{=}(\lambda u.fy)\ (fz)\ \mathsf{in}\ x]\!]\gamma') \\ \to & \underbrace{\exists\beta\gamma\beta'\gamma'\beta''\gamma''}_{=:\exists\ldots} \quad \underbrace{(\alpha{\doteq}\beta{\to}\gamma{\wedge}\gamma{\doteq}\beta'{\to}\gamma'{\wedge}\gamma'{\doteq}\beta''{\to}\gamma'')}_{=:\phi} \wedge \underbrace{y{:}\beta{\wedge}z{:}\beta'{\wedge}f{:}\beta''}_{=:\Gamma} {\wedge}[\![x]\!]\gamma'{\wedge}x{:}\delta/[\![(\lambda u.fy)\ (fz)]\!]\delta) \end{array}$$

Next, the abstraction encoding the polytype for x is simplified. Observe in particular, that the multistep transitions $\rightarrow^2$ and $\rightarrow^*$ introduce global variables (the types of f, y, and z into the abstraction body), along with their constraints according to congruence rule (PROP).

$$
\begin{array}{lllll}
 & \exists \ldots \phi \wedge \Gamma \wedge [\![x]\!]\gamma' \wedge x{:}\delta/(& & & [\![(\lambda u.fy)\,(fz)]\!]\delta\ ) \\
\rightarrow & \exists \ldots \phi \wedge \Gamma \wedge [\![x]\!]\gamma' \wedge x{:}\delta/(\exists\epsilon\zeta & \epsilon \doteq \zeta \rightarrow \delta \wedge & [\![(\lambda u.fy)]\!]\epsilon \wedge [\![(fz)]\!]\zeta\ ) \\
\rightarrow & \exists \ldots \phi \wedge \Gamma \wedge [\![x]\!]\gamma' \wedge x{:}\delta/(\exists\epsilon\epsilon'\zeta\zeta' & \epsilon \doteq \zeta \rightarrow \delta \wedge \epsilon' \doteq \zeta' \rightarrow \zeta & [\![(\lambda u.fy)]\!]\epsilon \wedge [\![f]\!]\epsilon' \wedge [\![z]\!]\zeta'\ ) \\
\rightarrow^2 & \exists \ldots \phi \wedge \Gamma \wedge [\![x]\!]\gamma' \wedge x{:}\delta/(\exists\epsilon\zeta & \epsilon \doteq \zeta \rightarrow \delta \wedge \beta'' \doteq \beta' \rightarrow \zeta & [\![(\lambda u.fy)]\!]\epsilon\ ) \\
\rightarrow & \exists \ldots \phi \wedge \Gamma \wedge [\![x]\!]\gamma' \wedge x{:}\delta/(\exists\epsilon\zeta\zeta'\kappa & \epsilon \doteq \zeta \rightarrow \delta \wedge \beta'' \doteq \beta' \rightarrow \zeta \wedge \epsilon \doteq \kappa \rightarrow \zeta' & u{:}\kappa \wedge [\![fy]\!]\zeta'\ ) \\
\rightarrow^* & \exists \ldots \phi \wedge \Gamma \wedge [\![x]\!]\gamma' \wedge x{:}\delta/(\exists\epsilon\zeta\zeta'\kappa & \epsilon \doteq \zeta \rightarrow \delta \wedge \beta'' \doteq \beta' \rightarrow \zeta \wedge \epsilon \doteq \kappa \rightarrow \zeta' \wedge \beta'' \doteq \beta \rightarrow \zeta' & u{:}\kappa\ ) \\
\equiv & \exists \ldots \phi \wedge \Gamma \wedge [\![x]\!]\gamma' \wedge x{:}\delta/(\exists\kappa & \beta'' \doteq \beta' \rightarrow \delta \wedge \beta' \doteq \beta & u{:}\kappa\ ) \\
\equiv & \exists \ldots \phi \wedge \Gamma \wedge [\![x]\!]\gamma' \wedge x{:}\delta/(& \beta'' \doteq \beta' \rightarrow \delta \wedge \beta' \doteq \beta &)
\end{array}
$$

Finally, we descend into the second branch of the let-expression, which means to apply the x-abstraction to get an instance of the polytype of x, and simplify the resulting constraint. The outcome is the principal type of $M : \beta \rightarrow \beta' \rightarrow \beta'' \rightarrow \gamma''$.

$$
\begin{array}{lllll}
 & \exists\beta\gamma\beta'\gamma'\beta''\gamma'' & (\phi & \wedge & y{:}\beta \wedge z{:}\beta' \wedge f{:}\beta'' \wedge [\![x]\!]\gamma' \wedge x{:}\delta/(\beta'' \doteq \beta' \rightarrow \delta \wedge \beta' \doteq \beta)) \\
\rightarrow & \exists\beta\gamma\beta'\gamma'\beta''\gamma'' & (\phi \wedge \beta'' \doteq \beta' \rightarrow \gamma' \wedge \beta' \doteq \beta & \wedge & y{:}\beta \wedge z{:}\beta' \wedge f{:}\beta'' \wedge x{:}\delta/(\beta'' \doteq \beta' \rightarrow \delta \wedge \beta' \doteq \beta)) \\
\equiv & \exists\beta\gamma\beta'\gamma'\beta''\gamma'' & (\phi \wedge \beta'' \doteq \beta' \rightarrow \gamma' \wedge \beta' \doteq \beta & \wedge & y{:}\beta \wedge z{:}\beta' \wedge f{:}\beta'' \wedge x{:}\delta/\beta'' \doteq \beta' \rightarrow \delta) \\
\equiv & \exists\beta\gamma\beta'\gamma'\beta''\gamma'' & (\phi \wedge \beta'' \doteq \beta' \rightarrow \gamma' \wedge \beta' \doteq \beta & \wedge & y{:}\beta \wedge z{:}\beta' \wedge f{:}\beta'') \\
\equiv & \exists\beta\beta'\beta''\gamma'' & (\alpha \doteq \beta \rightarrow \beta' \rightarrow \beta'' \rightarrow \gamma'') & &
\end{array}
$$

Acknowledgments

Thanks go to Gert Smolka for initiating my research on type checking and to Joachim Niehren for many fruitful discussions and joint research on this topic.

References

[1] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[2] M. Henz, M. Mehl, M. Müller, T. Müller, J. Niehren, R. Scheidhauer, C. Schulte, G. Smolka, R. Treinen, and J. Würtz. The Oz Handbook. DFKI, Saarbrücken, Germany, 1994. Available through anonymous ftp from `ps-ftp.dfki.uni-sb.de` or through www from `http://ps-www.dfki.uni-sb.de`.

[3] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[4] Martin Müller and Joachim Niehren. Higher-Order Meta Programming in Oz. Unpublished, 1994.

[5] Joachim Niehren. Personal Communication. 1994.

[6] Joachim Niehren. *Funktionale Berechnung in einem uniformen nebenläufigen Kalkül mit logischen Variablen*. PhD thesis, Universität des Saarlandes, 1994. forthcoming, in German.

[7] Joachim Niehren and Gert Smolka. A Confluent Calculus for Higher-Order Relational Programming. In *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, Munich, Germany, September 1994.

[8] Gert Smolka. A Foundation for Concurrent Constraint Programming. In *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, Munich, Germany, September 1994.

[9] Mitchell Wand. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae*, 10:115–122, 1987.