A Categorical Perspective on Pattern Unification (Extended Abstract)

Andrea Vezzosi and Andreas Abel

Department of Computer Science and Engineering
Chalmers University of Technology and Gothenburg University, Sweden
vezzosi@chalmers.se, andreas.abel@gu.se

Abstract

In 1991 Miller described a subset of the higher-order unification problem for the Simply Typed Lambda Calculus which admits most general unifiers, called the pattern fragment. This subset has been extended to more complex type theories and it is still used as the basis of modern unification algorithms in applications like proof search and type inference. Our contribution is a new presentation of the original unification algorithm that focuses on the abstract properties of the operations involved, using category theory as a structuring principle. These properties characterize a class of languages for which the algorithm can be reused.

1 Introduction

Pattern unification [Miller, 1991] is a restriction of higher-order unification where meta (unification) variables can only be applied to a list of distinct object (lambda bound) variables, called a *pattern*. This restriction is motivated by how such an unification problem with a meta variable at the head, $M\,x\,y\,z = t$, can essentially be read as a definition for the metavariable, $M := \lambda x\,y\,z.t$, as long as the resulting term is well-scoped.

The existence of most general unifiers guaranteed by the pattern restriction is important in applications like type inference for dependently typed languages or execution of higher order logic programs. In these cases, a common implementation strategy is to solve immediately those unification problems that fall into the pattern fragment and suspend the others, hoping that they will become tractable later when more meta variables have been solved [Reed, 2009, Abel and Pientka, 2011].

The basic intuition of our presentation, which is not new, is that patterns correspond to injective renamings and form a category; from there we go further and recognize how certain operations on patterns of the unification algorithm correspond to basic concepts of category theory like finite limits. The correctness of the resulting algorithm has been checked with a formalization [Vezzosi, 2012] using the proof assistant Agda. Category Theory has been used before to reason about first order unification by, for example, Goguen [Goguen, 1989].

2 The problem

We consider pattern unification in the Simply Typed Lambda Calculus (STLC) up to $\alpha\beta\eta$-equivalence. Without loss of generality, we can restrict our focus to terms in β-short η-long normal form by making use of the so-called spine formulation [Cervesato and Pfenning, 2003]. We use de Brujin [1972] indexes for object variables. As a consequence, unification is to be considered up to strict equality. Instead of a general application node $(t\,t)$ we have $(M\,p)$ and

$(x\,\vec{t})$ where the head is a meta or object variable. For application of meta variables $M\,p$ we represent the arguments by p, a list of distinct object variables, to ensure the pattern condition.

Terms	t	$::=$	λt	
		$\mid$	$x\,\vec{t}$	($x\,\vec{t}$ has base type ι)
		$\mid$	$M\,p$	($M\,p$ has base type ι)
Patterns	p	$::=$	$\vec{x}$	($x_i \neq x_j$ whenever $i \neq j$)
Object variables	x	$::=$	$0 \mid 1 + x$	

To match the way functions are always fully applied in terms, we define types in an uncurried style, i.e., as a (possibly empty) list of argument types and a base type ι for the result. We will simply write ι when there are no arguments.

Types	τ	$::=$	$\vec{\tau} \to \iota$	
Object contexts	Δ	$::=$	$\vec{\tau}$	
Meta contexts	Γ	$::=$	$. \mid M{:}\tau, \Gamma$	(Variables M in Γ are not repeated)

Typing contexts Δ for object variables are just lists of types. This identification between a typing context and argument types is exploited in the typing rules for metas and patterns. In the following, we list the rules for typing of variables $\Delta \vdash x : \tau$, patterns $\Delta \vdash p : \Delta'$, and normal terms $\Gamma; \Delta \vdash t : \tau$.

$$\frac{}{\tau, \Delta \vdash 0 : \tau} \qquad \frac{\Delta \vdash x : \tau}{\tau', \Delta \vdash 1 + x : \tau} \qquad \frac{}{\Delta \vdash . : .} \qquad \frac{\Delta \vdash x : \tau \qquad \Delta \vdash p : \Delta'}{\Delta \vdash (x, p) : (\tau, \Delta')}$$

$$\frac{\Gamma; \tau, \Delta \vdash t : \vec{\tau} \to \iota}{\Gamma; \Delta \vdash \lambda t : (\tau, \vec{\tau}) \to \iota} \qquad \frac{\Delta \vdash x : \Delta' \to \iota \qquad \Gamma; \Delta \vdash \vec{t} : \Delta'}{\Gamma; \Delta \vdash x\,\vec{t} : \iota} \qquad \frac{M : (\Delta' \to \iota) \in \Gamma \qquad \Delta \vdash p : \Delta'}{\Gamma; \Delta \vdash M\,p : \iota}$$

As hinted in the introduction we can think of $\Delta_2 \vdash p : \Delta_1$ as an injective renaming from variables in Δ_1 to variables in Δ_2, application to a variable $p\,x$ is performed by considering x from Δ_1 as an index to the position in p of the resulting variable in Δ_2. From this we form the category Pat with contexts as objects and patterns as morphisms, composition is given by $(p_1 \circ p_2)\,x = p_1\,(p_2\,x)$. We shall write $\Delta_2 \vdash p : \Delta_1$ as $p : \Delta_1 \to \Delta_2$.

We define *substitutions* σ as finite maps from meta variables to terms. *Update* $(\sigma, M := t)$ is defined as the substitution σ' such that $\sigma'\,M = t$ and $\sigma'\,N = \sigma\,N$ for $N \neq M$. Since we consider meta variables as living in a global scope, substitutions will produce terms without free object variables, hence they will be typed with an empty Δ. We write $\Gamma_2 \vdash \sigma : \Gamma_1$, or $\sigma : \Gamma_1 \to \Gamma_2$, iff $\sigma\,M = t$ for some $\Gamma_2; . \vdash t : \tau$ whenever $(M{:}\tau) \in \Gamma_1$.

Application of a substitution σ to a term, which we write $[\![\sigma]\!]\,t$, is done structurally as usual, except for nodes $(M\,p)$ where we need to normalize the generated beta-redex. Since p is merely a renaming, and our terms are η-long, normalizing amounts to stripping the outermost layer of λ abstractions from $(\sigma\,M)$ and applying p to their body. We do not need to apply σ to p because the latter does not contain meta variables.

$$[\![\sigma]\!]\,M\,p = [p]\,t \quad \text{where} \quad \sigma\,M = \vec{\lambda} t$$

In fact, from now on we will include the operation of stripping out the outer lambdas in the application of a substitution to a meta variable. Thus, given $\Gamma \vdash M : \Delta \to \iota$ we will have $\sigma\,M$ be a term t of type ι in the object context Δ, like the t in the equation above. This also allows us to express the *identity substitution* $\mathsf{id}_\Gamma : \Gamma \to \Gamma$ by simply $\mathsf{id}_\Gamma\,N = N\,\mathsf{id}_\Delta$ for $(N{:}\Delta \to \iota) \in \Gamma$. We write the *singleton substitution* $(\mathsf{id}, M := t)$ simply as $(M := t)$.

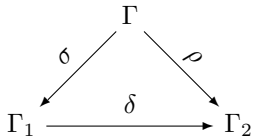
With the usual notion of composition we can also form the category Sub where meta variable contexts are the objects and substitutions are the morphisms.

Finally we observe that the set of terms with given type and contexts, $\mathsf{Tm}(\Gamma, \Delta, \tau)$, is functorial over both Sub and Pat, which is to say that application of renamings and substitutions commute with the respective compositions and identities, and between themselves. The category Type, is discrete, i.e., has types as objects but no morphisms except for identities.

$$\begin{array}{lcl} \mathsf{Tm} & : & \mathsf{Sub} \times \mathsf{Pat} \times \mathsf{Type} \to \mathsf{Set} \\ \mathsf{Tm}(\Gamma, \Delta, \tau) & = & \{t \in \mathrm{Terms} \mid \Gamma; \Delta \vdash t : \tau\} \end{array} \qquad \begin{array}{ll} [p_1 \circ p_2]\, t & = [p_1]\,[p_2]\, t \\ [\![\sigma_1 \circ \sigma_2]\!]\, t & = [\![\sigma_1]\!]\,[\![\sigma_2]\!]\, t \\ [\![\sigma]\!]\,[p]\, t & = [p]\,[\![\sigma]\!]\, t \end{array} \qquad \begin{array}{l} [\mathsf{id}_\Delta]\, t = t \\ [\![\mathsf{id}_\Gamma]\!]\, t = t \end{array}$$

Definition 1 (Unifier)**.** *A substitution $\sigma : \Gamma \to \Gamma_1$ is a* unifier *of two terms $t_1, t_2 \in \mathsf{Tm}(\Gamma, \Delta, \tau)$ whenever $[\![\sigma]\!]\, t_1 = [\![\sigma]\!]\, t_2$.*

Definition 2 (More General Substitution)**.** *A substitution $\sigma : \Gamma \to \Gamma_1$ is* more general *than $\rho : \Gamma \to \Gamma_2$, written $\sigma \leq \rho$, if there exists a substitution $\delta : \Gamma_1 \to \Gamma_2$ with $\rho = \delta \circ \sigma$.*



Definition 3 (Most General Unifier (MGU))**.** *A unifier $\sigma : \Gamma \to \Gamma_1$ of $t_1, t_2 \in \mathsf{Tm}(\Gamma, \Delta, \tau)$ is* most general *if $\sigma \leq \rho$ for every other unifier $\rho : \Gamma \to \Gamma_2$.*

3 Finding a solution

We will find the most general unifier of t_1 and t_2, or decide there cannot be one, by recursion on the terms themselves. In the following, we consider the possible cases.

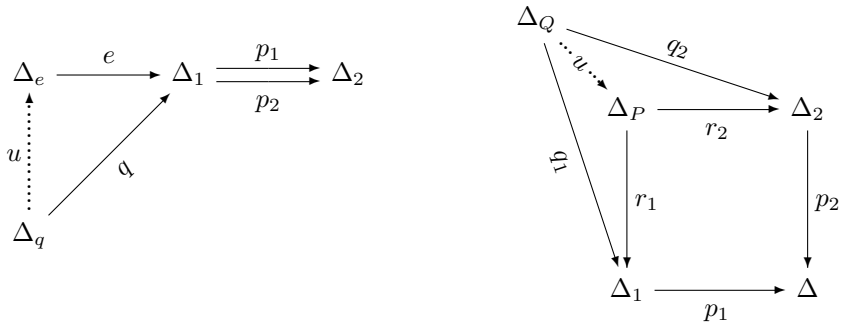
3.1 Rigid-Rigid

Since we have that $[\![\sigma]\!]\, \lambda t = \lambda([\![\sigma]\!]\, t)$, finding the unifier of λt_1 and λt_2 amounts to finding the one of t_1 and t_2. For variable applications, $x_1\, \vec{t_1}$ and $x_2\, \vec{t_2}$, it is almost the same, except that we need to check whether x_1 and x_2 are equal, and if so recurse over the subterms updating them with the unifier computed so far. In fact, we can abstract over both cases using a notion of operator $o ::= \lambda | x$ with decidable equality and arities, and such that $[\![\sigma]\!]\, (o\, \vec{t}) = o\, ([\![\sigma]\!]\, \vec{t})$.

3.2 Flex-Flex (Same Meta)

If the terms to unify are $M\, p_1$ and $M\, p_2$ things get more interesting. We can see that the most general unifier is $M := M'\, e$ where M' is a fresh meta variable and e is what is called the equalizer of p_1 and p_2. In fact, for σ to be an unifier it has to satisfy $[\![\sigma]\!]\, M\, p_1 = [\![\sigma]\!]\, M\, p_2$ which reduces to $M'\, (p_1 \circ e) = M'\, (p_2 \circ e)$ and the equalizer is the most general way to solve the equation $p_1 \circ e = p_2 \circ e$. What is meant by most general here is that for every other renaming q satisfying $p_1 \circ q = p_2 \circ q$ there is a unique u such that $q = e \circ u$. (See Figure 1.)

This property is all we need to show σ is most general, in fact, for any other unifier ρ we have $[p_1]\, (\rho\, M) = [p_2]\, (\rho\, M)$, and since the functor Tm preserves equalizers, i.e. $[e]$ is the equalizer of $[p_1]$ and $[p_2]$, we have a unique t such that $\rho\, M = [e]\, t$. That allows us to show $\sigma \leq \rho$ by $\delta := (\rho, M' := t)$. Now $(\delta \circ \sigma)\, M = [\![\delta]\!]\, M'\, e = [e]\, t = \rho\, M$ and $(\delta \circ \sigma)\, N = \delta\, N = \rho\, N$ for $N \neq M$.



Equalizer (Same Meta) Pullback (Different Meta)

Figure 1: Equalizer and Pullback diagrams

3.3 Flex-Flex (Different Meta)

As a stepping stone to the next case we consider the unification of the terms $M_1\, p_1$ and $M_2\, p_2$ for $M_1 \neq M_2$. In this case, the unifier is $\sigma = (M_1 := M'\, r_1, M_2 := M'\, r_2)$ for a fresh M' so that $[\![\sigma]\!]\, M_1\, p_1 = [\![\sigma]\!]\, M_2\, p_2$ reduces to $M_1\, (p_1 \circ r_1) = M_2\, (p_2 \circ r_2)$ and we can find r_1 and r_2 through another finite limit, the pullback of p_1 and p_2. (See Figure 1.)

3.4 Flex-Rigid

This is the main case to deal with, where we unify $M\, p$ with a t which does not contain M. Since $\sigma\, M$ will not be relevant for $[\![\sigma]\!]\, t$ we can decompose our candidate unifier σ into $(\pi, M := s)$ for a term s and another substitution π. The unification constraint $[\![\sigma]\!]\, M\, p = [\![\sigma]\!]\, t$ becomes $[p]\, s = [\![\pi]\!]\, t$. For this equation to be solvable the term $[\![\pi]\!]\, t$ may only have free object variables that appear in p. We distinguish free rigid variables, like x in $x\, \vec{t}$, from free pattern variables, like x in p. The substitution π can eliminate the free pattern variables which are not in p by so-called *pruning*.

Pruning. Pruning t with respect to p proceeds by recursion on t. In case $t = M'\, q$ we return the singleton substitution $\pi = (M' := M''\, r_2)$ for some fresh M'' and pattern r_2, so that $[\![\pi]\!]\, t = M''\, (q \circ r_2)$. The pattern $q \circ r_2$ must only contain free variables in p, which means there must exist a r_1 such that $p \circ r_1 = q \circ r_2$. The most general solution of this equation is again the pullback of p and q, and with an argument similar to the Flex-Flex case this leads to π being the most general pruning substitution. In case $t = \lambda t'$ we update the pattern p to handle the new bound variable, and recurse on t'. In case $t = x\, \vec{t}$ we recursively compute the pruning substitutions $\vec{\pi}$ for the subterms $\vec{t}$ and compose them by iteratively taking the pushout, i.e. the categorical dual of a pullback.

Inversion. Finding s such that $[p]\, s = [\![\pi]\!]\, t$, and thus uniquely solving the unification problem, is possible whenever the free rigid variables of t are contained in p. Specifically in the case $t = x\, \vec{t}$ the constraint reduces to $[p]\, s = x\, ([\![\pi]\!]\, \vec{t})$, this is solved by $s = y\, \vec{t_1}$ such that $p\, y = x$.

3.5 Failed Occurs Check

The remaining case is when we are tying to unify $M\,p$ with a term t where M appears in a nested position, e.g. $t = x\,(M\,q)$. Here we can conclude that there are no unifiers because the height of the two terms will never match.

4 Towards Generic Pattern Unification

Practical applications of higher-order unification need to handle languages more complex than STLC, e.g. with product and sum types, defined functions, and dependent types.

Our categorical view of the algorithm allows us to apply techniques from datatype generic programming to abstract away from the specific syntax and types of STLC. Instead of Pat we can consider a generic category Ctx having all the pullbacks and equalisers and whose arrows are monomorphisms. And instead of the grammar of STLC we can use an arbitrary one defined by a family of operators, as hinted in the Flex-Rigid section, as long as they are functorial with respect to Ctx, have decidable equality, and provide operations that attempt to invert the functorial action. From the functoriality of the operators we can derive the functoriality of the whole syntax. From a formalization point of view we would use an Indexed Container [Altenkirch and Morris, 2009] to describe the operators. It remains to be verified which language features of interest can fit into this generalization.

References

A. Abel and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In *Proc. of the 10th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2011*, volume 6690 of *Lect. Notes in Comput. Sci.*, pages 10–26. Springer, 2011.

T. Altenkirch and P. Morris. Indexed containers. In *Proc. of the 24nd IEEE Symp. on Logic in Computer Science (LICS 2009)*, pages 277–285. IEEE Computer Soc. Press, 2009.

I. Cervesato and F. Pfenning. A linear spine calculus. *J. Log. Comput.*, 13(5):639–688, 2003.

N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

J. A. Goguen. What is unification? – A categorical view of substitution, equation and solution. In *Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques*, pages 217–261. Academic, 1989.

D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.

J. Reed. Higher-order constraint simplification in dependent type theory. In *4th Int. Wksh. on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2009)*, pages 49–56. ACM Press, 2009.

A. Vezzosi. Higher-order pattern unification in Agda, 2012. URL http://github.com/Saizan/miller.