Higher-order unification from E-unification with second-order equations and parametrised metavariables

Nikolai Kudasov

Innopolis University, Tatarstan Republic, Russia
n.kudasov@innopolis.ru

Abstract

Type checking and, to a larger extent, type and term inference in programming languages and proof assistants can be implemented by means of unification. In the presence of dependent types, variations of higher-order unification are often used, such as higher-order pattern unification. In practice, there are several mechanisms for abstraction and application, as well as other eliminators (such as projections from pairs) that have to be accounted for in the implementation of higher-order unification for a particular language. In this work, we study the possibility of representing various higher-order unification problems as a special case of E-unification for a second-order algebra of terms. This allows us to present β-reduction rules for various application terms, and some other eliminators as equations, and reformulate higher-order unification problems as E-unification problems with second-order equations. We then present a procedure for deciding such problems, introducing a second-order *mutate* rule (inspired by one-sided paramodulation) and generic versions of the *imitate* and *project* rules. We also provide a prototype Haskell implementation for syntax-generic higher-order unification based on these ideas.

1 Introduction

Unification is often used as part of a type checking algorithm in programming languages and proof assistants. Unification also drives proof search through term inference algorithms in proof assistants and theorem provers.

In presence of dependent types, first-order unification becomes insufficient and variations of higher-order unification [10, 11, 5] are implemented. Mazzoli and Abel [13] describe how higher-order unification can be leveraged in a type checking algorithm for a dependently typed language. In general, higher-order pre-unification [10] and full unification [11] are only semi-decidable, and is rather expensive without non-trivial optimizations [16]. Instead, many dependent type inference algorithms choose to consider only some subset of unification problems that are sufficiently *nice*. For example, Miller's higher-order pattern unification [14] and its variations [9, 17] are often used as they present a decidable subset and still offer some convenience to the programmer, allowing them to omit some explicit type annotations.

In practice, many type theories offer several mechanisms for abstraction and application. For example, System F offers separate λ-abstraction for types and terms. Some calculi differentiate between call-by-name and call-by-value (e.g. $\lambda\mu\tilde{\mu}$-calculus and other sequent calculi [2]). Cubical type systems [3] introduce abstraction (and application) over the variable of the interval type $\mathbb{I}$. Riehl and Shulman's type theory with shapes [15] introduces extension types with a separate abstraction and application.

Additionally, higher-order unification is often extended to support eliminators other than function application. In particular, it is common to extend the pattern fragments to projections (for products and Σ-types) [1, 9]. This allows one to solve constraints where one of the terms is an application of a projection from a metavariable:

$$(\pi_2\ m_1)\ x\ y \overset{?}{=} \langle y, x\rangle \tag{1}$$

$$m_1 \mapsto \langle m_2, \lambda x_1.\lambda x_2.\langle x_2, x_1\rangle\rangle \tag{2}$$

In this work, we attempt to reformulate these different higher-order unification problems as a special case of a more general E-unification problem, and provide a single unification procedure that can be parameterised by the syntactic constructions of the object language and their evaluation rules.

Our approach is inspired by the one-sided paramodulation procedure [4, decomposition procedure], a simple technique used for equational unification (E-unification), that is unification of terms modulo a set E of first-order equalities. This technique assumes that E has a corresponding convergent abstract rewrite system R, which typically suits application to type-checking since reduction is typically provided for terms. We are most interested in the *mutate* rule (also known as *restructure* rule [4] and *lazy paramodulation* [8]):

$$\{F(e_1,\ldots,e_n) \overset{?}{=} t\} \cup S \Longrightarrow \{e_1 \overset{?}{=} l_1,\ldots,e_n \overset{?}{=} l_n, r \overset{?}{=} t\} \cup S$$
$$\text{when } F(l_1,\ldots,l_n) \longrightarrow r \text{ is a rewrite rule in } R \qquad \text{(mutate rule)}$$

The rule says that if the root symbol of term s in equation $s \overset{?}{=} t$ matches with one of the rewrite rules in R, we can decompose it into smaller equations using the rewrite rule.

Example 1.1. Consider the following rewrite rules:

$$\mathsf{first}(\mathsf{pair}(x,y)) \longrightarrow x \tag{3}$$

$$\mathsf{second}(\mathsf{pair}(x,y)) \longrightarrow y \tag{4}$$

Then, a constraint $\mathsf{second}(\mathsf{first}(p)) \overset{?}{=} t$ can be solved by applying the *mutate* rule as follows:

$$\{\mathsf{second}(\mathsf{first}(p)) \overset{?}{=} t\} \tag{5}$$
$$\Longrightarrow (\textit{mutate} \text{ rule with } \mathsf{second}(\mathsf{pair}(x,y)) \longrightarrow y)$$
$$\{\mathsf{first}(p) \overset{?}{=} \mathsf{pair}(x_1,y_1), t \overset{?}{=} y_1\} \tag{6}$$
$$\Longrightarrow (\textit{mutate} \text{ rule with } \mathsf{first}(\mathsf{pair}(x,y)) \longrightarrow x)$$
$$\{p \overset{?}{=} \mathsf{pair}(x_2,y_2), \mathsf{pair}(x_1,y_1) \overset{?}{=} x_2, t \overset{?}{=} y_1\} \tag{7}$$

These constraints can then be handled further by simplification and decomposition rules.

To get solutions for higher-order unification problems by means of E-unification, we require second-order equations. We will be using the second-order algebra of terms with parametrised metavariables by Fiore and Mahmoud [7, 6]. For instance, we will have a second-order rewrite rule corresponding to the β-reduction:

$$\mathsf{ap}(\mathsf{lam}_x(m_1[x]), m_2[]) \longrightarrow m_1[m_2[]] \tag{8}$$

The parametrised metavariables allow us to avoid explicit substitution in equations and are also helpful in the adaptation of the traditional *imitate* and *project* rules used in higher-order unification [10]. Those explicitly rely on "seeing" the bound variables and the head of a λ-term, but this information may not be fully available when we consider any specific metavariable locally without an explicit assumption that we have λ-abstraction. With parametrised metavariables, the *project* rule will select from a list of parameters of a metavariable instead.

Our expected contributions are as follows:

1. We formulate a notion of an E^2-unification problem (for a lack of a better name) for a set E of second-order equations with parameterised metavariables. We also show how higher-order unification can be seen as a special case of E^2-unification.
2. We describe a basic non-deterministic pre-unification[1] procedure for solving E^2-unification problems, with special versions of the *mutate*, *imitate*, and *project* rules.
3. We prove the pre-unification procedure to be complete and semi-decidable.
4. We present a prototype implementation of the syntax-generic higher-order unification procedure in the Haskell programming language based on these ideas [12].

2 E^2-unification problem

For the rest of this note, let M be a set of metavariable names. We start by formally introducing second-order terms with fully applied metavariables:

Definition 2.1 (terms,equations,rewrite rules,substitution,metasubstitution)**.** Let C be a set of function symbols. The set $T_C(X)$ of **second-order terms with fully applied metavariables** over the set X of (free) variables is defined recursively as follows:

1. $x \in T_C(X)$ when $x \in X$ (x is a variable);
2. $m[t_1, \ldots, t_n] \in T_C(X)$ when $m \in M$ (m is a metavariable) and for each $1 \leq i \leq n$ we have $t_i \in T_C(X)$;
3. $F(t_1, \ldots, t_n) \in T_C(X)$ when $F \in C$ (F is a function symbol) and for each $1 \leq i \leq n$ either $t_i \in T_C(X)$ or $t_i = z.s_i$ (where $s_i \in T_C(\{z\} \cup X)$).

Equations $m_1[k_1], \ldots, m_n[k_n] \vdash t_1 \equiv t_2$ and **rewrite rules** $m_1[k_1], \ldots, m_n[k_n] \vdash t_1 \longrightarrow t_2$ are simply pairs of terms in the context of metavariables $m_1, \ldots, m_n$ with arities $k_1, \ldots, k_n$. Given a term $t \in T_C(X)$, variables $x, x_1, \ldots, x_k \notin X$, and metavariable $m \in M$ we define **substitution** $[x \mapsto t]$ and **metasubstitution** $[m[x_1, \ldots, x_k] \mapsto t]$ in the usual way [6, 7].

Definition 2.2 (constraint,E^2-unification problem)**.** Let E be a set of equations of second-order terms with fully applied metavariables. Let t_1, t_2 be terms. Then $c = \forall z_1, \ldots, z_k.t_1 \overset{?}{=} t_2$ is a **constraint**. A tuple (C, E, Cs) where C is a set of function symbols, X is a set of variables, E is a set of equations over $T_C(\varnothing)$, Cs is a set of constraints over $T_C(X)$, is called an **E^2-unification problem**. A metavariable substitution σ is called a **solution to the E^2-unification problem** (C, E, Cs) if σCs contains only trivial constraints $\forall \overline{z_k}.t_1 = t_2$ where $t_1 = t_2$ modulo E.

Example 2.3 (untyped lambda calculus)**.** For the untyped λ-calculus, we set $C_\lambda = \{\mathsf{ap}, \mathsf{lam}\}$. Well-formed λ-terms will be a subset of $T_{C_\lambda}(X)$. Closed λ-terms will be a subset of T_{C_λ}. We set the equations E_λ corresponding to a single rewrite rule (β-reduction rule):

$$\mathsf{ap}(\mathsf{lam}(x.m_1[]), m_2[]) \longrightarrow m_1[m_2[]] \tag{9}$$

Consider the following constraint:

$$\mathsf{ap}(\mathsf{ap}(m_3[y], f), x) \overset{?}{=} \mathsf{ap}(\mathsf{ap}(f, y), x) \tag{10}$$

[1]pre-unification procedure solves rigid-rigid and flex-rigid constraints, but does not solve flex-flex constraints (i.e. when terms on both sides are metavariables)

The following metavariable substitution constitutes a solution:

$$[m_3[z_1] \mapsto \mathsf{lam}(z_2.\mathsf{ap}(z_1, z_2))])] \tag{11}$$

Higher-order unification problem for untyped λ-calculus corresponds directly to the E^2_λ-unification problem for well-formed λ-terms.

3 The unification procedure

To solve E^2-unification problems we propose a non-deterministic procedure specified by the following rules. Each rule takes a set of constraints as input and maps it onto a new set of constraints together with a (potentially empty) metavariable substitution.

3.1 Decomposition

The three rules below can be collectively described as *decomposition rules*, since they do not introduce any metavariable substitutions and instead only remove or break down some of the constraints.

Delete rule. This rule simply removes a constraint where both sides are equal:

$$S \uplus \{\forall \overline{z_k}.t \overset{?}{=} t\} \mapsto \langle S, []\rangle \tag{delete}$$

Simplify rule. This rule breaks down a constraint where both sides have the same function symbol at the root:

$$S \uplus \{\forall \overline{z_k}.F(\overline{t_n}) \overset{?}{=} F(\overline{u_n})\} \mapsto \langle S \uplus \{\forall \overline{z_k}.t_1 \overset{?}{=} u_1, \ldots, \forall \overline{z_k}.t_n \overset{?}{=} u_n\}, []\rangle \tag{simplify}$$

Technically, some of the subterms t_i, u_i can have the form $x.s_i$ and $y.v_i$ (introducing scope). In this case, the notation $\forall \overline{z_k}.\ x.s_i \overset{?}{=} y.v_i$ is understood as $\forall \overline{z_{k+1}}.[x \mapsto z_{k+1}]s_i \overset{?}{=} [y \mapsto z_{k+1}]v_i$ (assuming z_{k+1} is *fresh*).

Mutate rule. This rule essentially aims to unify one of the terms in the constraint with the left hand side of some rewrite rule:

$$S \uplus \{\forall \overline{z_k}.F(\overline{t_n}) \overset{?}{=} u\} \mapsto \langle S \uplus \{\forall \overline{z_k}.t_1 \overset{?}{=} l'_1, \ldots \forall \overline{z_k}.t_n \overset{?}{=} l'_n, \forall \overline{z_k}.r' \overset{?}{=} u\}, []\rangle \tag{mutate}$$
$$\text{when } F(\overline{l_k}) \longrightarrow u \text{ is in } R$$

In this rule, all parameters of the rewrite rule are enriched with extra parameters $\overline{z_k}$. For example, in the rule $\pi_1(\mathsf{mkPair}(m_1[], m_2[])) \longrightarrow m_1[]$, we have $l'_1 = \mathsf{mkPair}(m_1[\overline{z_k}, m_2[\overline{z_k}])]$ and $r' = m_1[\overline{z_k}]$.

3.2 Construction rules

The role of the remaining rules is to construct *candidate solutions* for constraints where one of the terms is just a metavariable. The main idea follows the original idea of Huet [10]. We can either *imitate* the other term (i.e. build a term from the "head" of the other term), or *project* (i.e. build a term from one of the available arguments of the metavariable). However, before we can state these two rules, we need to define how we "build a candidate solution" from a starting seed.

Definition 3.1 (candidate shapes)**.** Let T_C be a family of terms, R be a set of rewrite rules over T_C, and $H \subseteq T_C(\overline{z_k})$ be a set of terms, then the set $\mathsf{Shapes}_{\overline{z_k}}(H)$ of **candidate shapes** is defined recursively as follows:

1. $t \in \mathsf{Shapes}_{\overline{z_k}}(H)$ when $t \in H$;
2. $F(\overline{t_n}) \in \mathsf{Shapes}_{\overline{z_k}}(H)$ when $F(\overline{l_n}) \longrightarrow r$ is in R and for each $1 \leq i \leq n$
   (a) if l_i is a metavariable, then $t_i = m_i[\overline{z_k}]$, where m_i is a fresh metavariable;
   (b) if $l_i = x.s_i$ then $t_i = z_{k+1}.u_i$ and $u_i \in \mathsf{Shapes}_{\overline{z_{k+1}}}(H)$
   (c) otherwise $t_i \in \mathsf{Shapes}_{\overline{z_k}}(H)$

We also need to define how to extract the head(s) from a term, since we are not assuming any particular structure of the terms:

Definition 3.2 (head subterms, heads of a term)**.** Let $T : \mathsf{Set} \to \mathsf{Set}$ be a family of terms, and R be a set of rewrite rules over T. A subterm t_i of a term $F(\overline{t_n})$ is called a **head subterm** if there is a rewrite rule $F(\overline{l_n}) \longrightarrow r$ in R such that l_i is not a metavariable $m_i[\ldots]$ or a scoped meta variable $x.m_i[\ldots]$. Set $\mathsf{heads}(t)$ of **heads of a term** t is defined recursively as follows:

1. $u \in \mathsf{heads}(t)$ if $u \in \mathsf{heads}(s)$ and s is a head subterm of t;
2. $t \in \mathsf{heads}(t)$ if t has no head subterms.

Imitate/project rule. This rule constructs a metavariable substitution by selecting a shape built from a set of heads of the other term (*imitate*) or from a set of arguments passed to the metavariable (*project*):

$$\begin{aligned} S \uplus \{\forall \overline{z_k}.m[\overline{t_n}] \overset{?}{=} u\} &\mapsto \langle \sigma S, \sigma \rangle && \text{(imitate/project)} \\ &\text{when } \sigma = [m[\overline{x_n}] \mapsto s] \\ &\text{and } s \in \mathsf{Shapes}_{\overline{z_k}}(\mathsf{heads}(u) \cup \{x_1, \ldots, x_n\}) \end{aligned}$$

4 Implementation

We have implemented syntax-generic higher-order unification based on ideas similar to this paper in Haskell [12]. There, we use a particular presentation of syntax of terms using a combination of techniques from free monads and intrinsic scoping (via de Bruijn indices as nested data types).

References

[1] Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In C.-H. Luke Ong, editor, *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 10–26. Springer, 2011. doi: 10.1007/978-3-642-21691-6_5. URL https://doi.org/10.1007/978-3-642-21691-6_5.

[2] Zena M. Ariola, Aaron Bohannon, and Amr Sabry. Sequent calculi and abstract machines. *ACM Trans. Program. Lang. Syst.*, 31(4):13:1–13:48, 2009. doi: 10.1145/1516507.1516508. URL https://doi.org/10.1145/1516507.1516508.

[3] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017. URL http://collegepublications.co.uk/ifcolog/?00019.

[4] Nachum Dershowitz and G. Sivakumar. Solving goals in equational languages. In Stéphane Kaplan and Jean-Pierre Jouannaud, editors, *Conditional Term Rewriting Systems, 1st International Workshop, Orsay, France, July 8-10, 1987, Proceedings*, volume 308 of *Lecture Notes in Computer Science*, pages 45–55. Springer, 1987. doi: 10.1007/3-540-19242-5_4. URL https://doi.org/10.1007/3-540-19242-5_4.

[5] Conal Elliott. Higher-order unification with dependent function types. In Nachum Dershowitz, editor, *Rewriting Techniques and Applications, 3rd International Conference, RTA-89, Chapel Hill, North Carolina, USA, April 3-5, 1989, Proceedings*, volume 355 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 1989. doi: 10.1007/3-540-51081-8_104. URL https://doi.org/10.1007/3-540-51081-8_104.

[6] Marcelo P. Fiore and Chung-Kil Hur. Second-order equational logic (extended abstract). In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2010. doi: 10.1007/978-3-642-15205-4_26. URL https://doi.org/10.1007/978-3-642-15205-4_26.

[7] Marcelo P. Fiore and Ola Mahmoud. Second-order algebraic theories. *CoRR*, abs/1308.5409, 2013. URL http://arxiv.org/abs/1308.5409.

[8] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general e-unification. *Theor. Comput. Sci.*, 67(2&3):203–260, 1989. doi: 10.1016/0304-3975(89)90004-2. URL https://doi.org/10.1016/0304-3975(89)90004-2.

[9] Adam Michael Gundry. *Type inference, Haskell and dependent types*. PhD thesis, University of Strathclyde, Glasgow, UK, 2013. URL http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22728.

[10] Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975. doi: 10.1016/0304-3975(75)90011-0. URL https://doi.org/10.1016/0304-3975(75)90011-0.

[11] D. C. Jensen and Tomasz Pietrzykowski. Mechanizing *omega*-order type theory through unification. *Theor. Comput. Sci.*, 3(2):123–171, 1976. doi: 10.1016/0304-3975(76)90021-9. URL https://doi.org/10.1016/0304-3975(76)90021-9.

[12] Nikolai Kudasov. Functional pearl: Dependent type inference via free higher-order unification, 2022. URL https://arxiv.org/abs/2204.07454.

[13] Francesco Mazzoli and Andreas Abel. Type checking through unification. *CoRR*, abs/1609.09709, 2016. URL http://arxiv.org/abs/1609.09709.

[14] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991. doi: 10.1093/logcom/1.4.497. URL https://doi.org/10.1093/logcom/1.4.497.

[15] Emily Riehl and Michael Shulman. A type theory for synthetic ∞-categories. *Higher Structures*, 1, 2017.

[16] Petar Vukmirovic, Alexander Bentkamp, and Visa Nummelin. Efficient full higher-order unification. *Log. Methods Comput. Sci.*, 17(4), 2021. doi: 10.46298/lmcs-17(4:18)2021. URL https://doi.org/10.46298/lmcs-17(4:18)2021.

[17] Beta Ziliani and Matthieu Sozeau. A unification algorithm for coq featuring universe polymorphism and overloading. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 179–191. ACM, 2015. doi: 10.1145/2784731.2784751. URL https://doi.org/10.1145/2784731.2784751.