Generalising Huet-style Projections in E-unification for Second-Order Abstract Syntax

Nikolai Kudasov

Generalising Huet-style Projections in E-unification for Second-Order Abstract Syntax

Nikolai Kudasov

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

Abstract

Second-order abstract syntax (SOAS) provides a well-founded framework when working with languages that have arbitrary binders. E-unification for SOAS is powerful enough to encode both E-unification and higher-order unification (HOU). In a previous work, we have presented an E-unification procedure, drawing inspiration from both E-unification and HOU algorithms. In this paper, we present a work-in-progress on an more efficient version of the procedure, aiming to bring E-unification for SOAS to perform on par with HOU algorithms. More specifically, in this work we focus on a generalisation of Huet-style projections (also known as *partial bindings* in HOU literature), commonly used in HOU algorithms.

Contents

1 Introduction

Second-order abstract syntax (SOAS) [3] offers a mathematically sound approach to work with languages with binders. SOAS is an abstract syntax with variable bindings and parameterised metavariables. Fiore and Szamozvancev [2] have successfully used SOAS to generate metatheory in Agda for user-defined languages with variable bindings. SOAS has also been used by Makoto Hamana in his studies of second-order rewriting and Second-Order Laboratory [5].

In a previous work, the author [9, 10] has introduced E-unification for second-order abstract syntax, combining approaches from first-order E-unification [4] and higher-order unification (HOU) [7, 13]. Although the algorithm is sound and complete, it is not particularly efficient and several directions for improvements are outlined in the papers.

In another work, the author [8] has implemented a typechecking algorithm based on a (simplified) second-order abstract syntax. One of the heuristics used in that implementation resembles generalised Huet-style project bindings for untyped syntax.

$$\frac{x : \tau \in \Gamma}{\Theta \mid \Gamma \vdash x : \tau}\ \text{variables} \qquad \frac{\mathsf{M} : [\sigma_1, \ldots, \sigma_n]\tau \in \Theta \quad \text{for all } i = 1, \ldots, n \quad \Theta \mid \Gamma \vdash t_i : \sigma_i}{\Theta \mid \Gamma \vdash \mathsf{M}[t_1, \ldots, t_n] : \tau}\ \text{metavariables}$$

$$\frac{\mathsf{F} : (\overline{\sigma_1}.\tau_1, \ldots, \overline{\sigma_n}.\tau_n) \to \tau \quad \text{for all } i = 1, \ldots, n \quad \Theta \mid \Gamma, \overline{x_i} : \overline{\sigma_i} \vdash t_i : \tau_i}{\Theta \mid \Gamma \vdash \mathsf{F}(\overline{x_1}.t_1, \ldots, \overline{x_n}.t_n) : \tau}\ \text{operators}$$

Figure 1: Second-order terms in context.

In this paper, we demonstrate a work-in-progress for improving the efficiency of E-unification procedure for SOAS, by attempting to generalise Huet-style projection bindings (used in many HOU algorithms [6, 12, 1, 11, 13]) in the simply-typed setting, use them to improve **(project)**, **(imitate)**, and **(iterate)** transition rules [10, Section 4.2] and cut down the search space for the unification procedure [10, Definition 33].

2 E-unification for Second-Order Abstract Syntax

2.1 Second-Order Abstract Syntax

Second-order abstract syntax [3, Section 2] consists of

1. A signature Σ, defining the set of types T and available operators. Each operator F has an *arity* $(\overline{\sigma_1}.\tau_1, \ldots, \overline{\sigma_n}.\tau_n) \to \tau$, that specifies a return type τ and, for each argument, types of bound variables $\overline{\sigma_i}$ introduced in that argument, and the type τ_i of the argument itself.
2. Typing contexts $\Xi \mid \Gamma$, where Ξ defines the types of parametrised metavariables and Γ specifies the types of regular variables. For each metavariable typing $\mathsf{M} : [\overline{\sigma}]\tau$, we specify the types of all parameters $\overline{\sigma}$ as well as the return type τ.
3. Terms that are given by the rules in Fig. 1.

An ***equational presentation*** [3, Section 5] is a set of axioms each of which is a pair of terms in context.

Example 1. *A signature for simply-typed λ-calculus with pairs over a set of base types A consists of:*

1. *a set of types $A^{\Rightarrow,\times}$ given by*

$$\frac{\tau \in A}{\tau \in A^{\Rightarrow,\times}} \qquad \frac{\sigma, \tau \in A^{\Rightarrow,\times}}{\sigma \Rightarrow \tau \in A^{\Rightarrow,\times}} \qquad \frac{\sigma, \tau \in A^{\Rightarrow,\times}}{\sigma \times \tau \in A^{\Rightarrow,\times}}$$

2. *a family of operators for all* $\sigma, \tau \in A^{\Rightarrow,\times}$

$$\begin{aligned}
\mathsf{abs}^{\sigma,\tau} &: \sigma.\tau \to (\sigma \Rightarrow \tau) && \text{(abstraction)}\\
\mathsf{app}^{\sigma,\tau} &: (\sigma \Rightarrow \tau, \sigma) \to \tau && \text{(application)}\\
\mathsf{pair}^{\sigma,\tau} &: (\sigma, \tau) \to \sigma \times \tau && \text{(pair constructor)}\\
\mathsf{fst}^{\sigma,\tau} &: \sigma \times \tau \to \sigma && \text{(first projection)}\\
\mathsf{snd}^{\sigma,\tau} &: \sigma \times \tau \to \tau && \text{(second projection)}
\end{aligned}$$

Example 2. *The equational presentation of the simply-typed* λ*-calculus with pairs:*

$$\begin{aligned}
\mathrm{M} : [\sigma]\tau, \mathrm{N} : []\sigma \mid \cdot \vdash \mathsf{app}(\mathsf{abs}(x.\mathrm{M}[x]), \mathrm{N}[]) &\equiv \mathrm{M}[\mathrm{N}[]] : \tau && (\beta_\lambda)\\
\mathrm{M} : []\sigma, \mathrm{N} : []\tau \mid \cdot \vdash \mathsf{fst}(\mathsf{pair}(\mathrm{M}[], \mathrm{N}[])) &\equiv \mathrm{M}[] : \sigma && (\beta_{\pi_1})\\
\mathrm{M} : []\sigma, \mathrm{N} : []\tau \mid \cdot \vdash \mathsf{snd}(\mathsf{pair}(\mathrm{M}[], \mathrm{N}[])) &\equiv \mathrm{N}[] : \tau && (\beta_{\pi_2})\\
\mathrm{M} : []\sigma \Rightarrow \tau \mid \cdot \vdash \mathsf{abs}(x.\mathsf{app}(\mathrm{M}[], x)) &\equiv \mathrm{M}[] : \sigma \Rightarrow \tau && (\eta_\lambda)\\
\mathrm{M} : []\sigma \times \tau \mid \cdot \vdash \mathsf{pair}(\mathsf{fst}(\mathrm{M}[]), \mathsf{snd}(\mathrm{M}[])) &\equiv \mathrm{M}[] : \sigma \times \tau && (\eta_\times)
\end{aligned}$$

2.2 E-unification procedure for SOAS

E-unification problems for SOAS are about finding metavariable substitutions satisfying a collection of second-order constraints (i.e. equalities of second-order terms):

Definition 1 ([10, Definition 8])**.** *Given an equational presentation* E*, an* E**-unification problem** $\langle \Theta, S \rangle$ *is a finite set* S *of second-order constraints in a shared metavariable context* Θ*. We present an* E*-unification problem as a formula of the following form:*

$$\exists(\mathrm{M}_1 : [\overline{\sigma_1}]\tau_1, \ldots, \mathrm{M}_n : [\overline{\sigma_n}]\tau_n).(\forall(\overline{z_1} : \overline{\rho_1}).s_1 \stackrel{?}{=} t_1 : \tau_1) \wedge \ldots \wedge (\forall(\overline{z_k} : \overline{\rho_k}).s_k \stackrel{?}{=} t_k : \tau_k)$$

Here is an example of E-unification problem for simply-typed λ-terms:

$$\exists(\mathrm{M} : [\sigma \Rightarrow \tau, (\sigma \Rightarrow \tau) \Rightarrow \tau]\tau).\ \forall(g : \sigma \Rightarrow \tau, y : \sigma).\ \mathrm{M}[g, \mathsf{abs}(x.\mathsf{app}(x, y))] \stackrel{?}{=} \mathsf{app}(g, y) : \tau$$

An E-unification procedure introduced in a previous work [10] consists of a collection of transition rules that are applied non-deterministically:

Definition 2 ([10, Definition 33])**.** *The* E**-unification procedure** *over an equational presentation* E *is defined by repeatedly applying the following transitions (non-deterministically) until a stop:*

1. *If no constraints are left, then stop* ***(succeed)****.*
2. *If possible, apply* ***(delete)*** *rule.*
3. *If possible, apply* ***(mutate)*** *or* ***(decompose)*** *rule (non-det.).*
4. *If there is a constraint consisting of two non-metavariables and none of the above transitions apply, stop* ***(fail)****.*
5. *If there is a constraint* $\mathrm{M}[\ldots] \stackrel{?}{=} \mathsf{F}(\ldots)$*, apply* ***(imitate)*** *or* ***(project)*** *rules (non-det.).*
6. *If there is a constraint* $\mathrm{M}[\ldots] \stackrel{?}{=} x$*, apply* ***(project)*** *rules (non-det.).*
7. *If possible, apply* ***(identify)****,* ***(eliminate)****, or* ***(iterate)*** *rules (non-det.).*

8. If none of the rules above are applicable, then stop ***(fail)****.*

We refer the reader to [10, Section 4] for the full description and examples of the transition rules. Importantly, all of these rules rely on exact matching with axioms (in particular, computation rules of a given language), use type information superficially, resulting in a small-step advancements towards a solution. For example, constraint $\mathsf{app}(\mathsf{app}(\mathtt{M}[], t_1), t_2) \stackrel{?}{=} r$ will perform two **(mutate)** rules to first figure out substitution $\mathtt{M}[] \mapsto \mathsf{abs}(x_1.\mathsf{abs}(x_2.\mathtt{M}_2[x_1, x_2]))$ and get to solving $\mathtt{M}_2[t_1, t_2] \stackrel{?}{=} r$. Furthermore, without relying on the types of t_1, t_2, and r parameters, the procedure has to fallback to trial and error in figuring out how to use t_1, t_2 to produce r. Huet-style projection [6], explicitly avoided in the these transition rules, relies on the types of the arguments of a metavariable, to properly eliminate them in a single step. Instead, the procedure relies only on Jensen-Pietrzykowski-style (JP-style) [7], which simply selects the parameter to be used, but defers proper elimination (e.g. application to arguments) to other parts of the procedure.

3 Generalising Huet-style projection

3.1 Huet-style projection

We start with a formal definition of Huet-style projection bindings for first-order syntax:

Definition 3. *Let* $\mathtt{M} : \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta$ *and for some* i, $\alpha_i = \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_m \Rightarrow \beta$ *such that* $n > 0$ *and* $m \geq 0$*. Then a* **first-order Huet-style projection binding** *is a substitution*

$$\mathtt{M} \mapsto \lambda \overline{x}.x_i \ (\mathtt{H}_1 \ \overline{x}) \ \ldots \ (\mathtt{H}_m \ \overline{x})$$

where the metavariables $\overline{\mathtt{H}}$ *are fresh and bound variables* $\overline{x}$ *are of appropriate types (note that* i *in* x_i *and* α_i *is the same).*

In a higher-order unification procedure, this binding can be used whenever we want to use one of the parameters of a metavariable directly to produce the desired term. For example, consider the following unification problem:

$$\begin{aligned}
&\mathtt{M} \ (\lambda x.x \ y) \ f \stackrel{?}{=} f \ y \\
\xrightarrow{\text{H-project}} \ &(\lambda x_1 \ x_2.x_1 \ (\mathtt{H}_1 \ x_1 \ x_2)) \ (\lambda x.x \ y) \ f \stackrel{?}{=} f \ y \qquad \text{via} \quad [\mathtt{M} \mapsto \lambda x_1 \ x_2.x_1 \ (\mathtt{H}_1 \ x_1 \ x_2)] \\
\longrightarrow_\beta^* \ &(\mathtt{H}_1 \ (\lambda x.x \ y) \ f) \ y \stackrel{?}{=} f \ y \\
\xrightarrow{\text{decompose}} \ &(\mathtt{H}_1 \ (\lambda x.x \ y) \ f) \stackrel{?}{=} f \\
\xrightarrow{\text{JP-project}} \ &((\lambda x_1 \ x_2.x_2) \ (\lambda x.x \ y) \ f) \stackrel{?}{=} f \qquad \text{via} \quad [\mathtt{H}_1 \mapsto \lambda x_1 \ x_2.x_2] \\
\longrightarrow_\beta^* \ &f \stackrel{?}{=} f
\end{aligned}$$

Let us recognise the following components of Definition 3:

1. The type of i-th parameter (α_i) determines if this parameter can be used to extract the result of type β. Note that $\gamma_1 \Rightarrow \gamma_2 \Rightarrow \beta = \gamma_1 \Rightarrow (\gamma_2 \Rightarrow \beta)$, which means, that we may need to go deep into the structure of the type α_i to understand if it contains the type β, moreover, if β appears in a position that is suitable for extraction (a positive position for function types).

2. The type of $\mathtt{M}$ specifies the parameters as well as the return type. The usual assumption is that β is not itself a function type. We use the parameter types to *introduce* as many parameters as needed using λ-abstration.
3. The structure of the type of i-th parameter tells us how to *eliminate* it using a proper number of arguments to extract a term of type β.
4. Each argument used in the elimination of $x_i : \alpha_i$ is a fresh metavariable with the same parameters as the original metavariable. The arguments are added using function application (represented by juxtaposition) which is the eliminator for function types.

3.2 Adding product types

Let us add product types, pairs of terms, and pair projections to the simply typed λ-calculus:

1. For all types σ, τ, we add the product type $\sigma \times \tau$.
2. If $t_1 : \sigma$ and $t_2 : \tau$ are terms, then $\langle t_1, t_2 \rangle : \sigma \times \tau$ is a pair.
3. If $t : \sigma \times \tau$, then $\pi_1\ t : \sigma$ and $\pi_2\ t : \tau$ are the first and second projections, respectively.

Consider the constraint

$$\mathtt{M}\ \langle \lambda x.x\ y, f \rangle \stackrel{?}{=} f\ y$$

Assuming $f : \sigma \Rightarrow \tau$ and $y : \sigma$, the type of the only parameter to $\mathtt{M}$ is $((\sigma \Rightarrow \tau) \Rightarrow \tau) \times (\sigma \Rightarrow \tau)$. It is clear that we should can extract both components with π_1 or π_2 projections. In particular, a binding that will lead to a unifier for this constraint looks as follows:

$$\mathtt{M} \mapsto \lambda x_1.\pi_1\ x_1\ (\mathtt{H}_1\ x_1)$$

We observe the following from this example:

1. The type of a parameter to $\mathtt{M}$ determines whether it can be used to extract the result of appropriate type.
2. The structure of the parameter type tells us how to *eliminate* it. However, unlike the case of function types, we now have two eliminators: π_1 and π_2.

To generalise Huet-style projection bindings to include product types, we define the set of possible formulas on the right hand side recursively over the type of i-th parameter of the metavariable.

1. $\mathsf{rhs}_{\overline{x},\beta}(t, \beta) = \{t\}$
2. $\mathsf{rhs}_{\overline{x},\beta}(t, \delta) = \varnothing$ when $\delta \neq \beta$ and δ is not a function or product type
3. $\mathsf{rhs}_{\overline{x},\beta}(t, \gamma \Rightarrow \delta) = \mathsf{rhs}_{\overline{x},\beta}(t\ (\mathtt{H}_1\ \overline{x}), \delta)$ where $\mathtt{H}_1 : \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \gamma$ is fresh
4. $\mathsf{rhs}_{\overline{x},\beta}(t, \delta_1 \times \delta_2) = \mathsf{rhs}_{\overline{x},\beta}(\pi_1\ t, \delta_1) \cup \mathsf{rhs}_{\overline{x},\beta}(\pi_2\ t, \delta_2)$

The set of possible Huet-style projection bindings for metavariable $\mathtt{M} : \alpha_1 \Rightarrow \ldots \alpha_n \Rightarrow \beta$ at position i is defined as

$$\{[\mathtt{M} \mapsto \lambda \overline{x}.s] \mid s \in \mathsf{rhs}_{\overline{x},\beta}(x_i, \alpha_i)\}$$

The cases 3 and 4 above stem from the canonical values for function and product types and the following computation rules for simply-typed λ-calculus with pairs:

$$(\lambda x.\mathtt{M})\ \mathtt{N} \longrightarrow [x \mapsto \mathtt{N}]\mathtt{M} \qquad (\beta_\lambda)$$
$$\pi_1\ \langle \mathtt{M}, \mathtt{N} \rangle \longrightarrow \mathtt{M} \qquad (\beta_{\pi_1})$$
$$\pi_2\ \langle \mathtt{M}, \mathtt{N} \rangle \longrightarrow \mathtt{N} \qquad (\beta_{\pi_2})$$

This suggests that a generalisation of Huet-style bindings is possible based on the set of type constructors and the set of axioms E as given in Example 2.

3.3 Generalising to SOAS

We now consider a generalisation of Huet-style projection bindings for an arbitrary second-order abstract syntax. First, assume that $\mathtt{M} : [\alpha_1, \dots, \alpha_n]\beta$. Let $x : \alpha_1, \dots, x_n : \alpha_n$. Then we define the set of right-hand side formulas $\mathsf{rhs}_{\overline{x},\beta}$ recursively as follows:

1. $\mathsf{rhs}_{\overline{x},\beta}(t, \beta) = \{t\}$
2. $\mathsf{rhs}_{\overline{x},\beta}(t, \delta) = \bigcup_{\langle l,\xi\rangle \in L} \mathsf{rhs}_{\overline{x},\beta}([y \mapsto t]\xi l, \varepsilon)$
   where $L = \{\langle l, \xi\rangle \mid \exists(\Theta \mid \cdot \vdash s : \delta).(\Theta \mid \cdot \vdash [y \mapsto s]l \equiv r : \varepsilon) \in E\}$
   and each $l \neq y$ and each ξ instantiates l in the appropriate context by substituting each metavariable with a fresh one
3. $\mathsf{rhs}_{\overline{x},\beta}(t, \delta) = \varnothing$ otherwise

Intuitively, the set L in case 2 specifies all possible contexts where we could put a term of type δ such that one of the axioms (e.g. computation rules) will fire. Note that we are not looking at the term t to find such contexts, only its type. For example, if $\mathit{delta} = \sigma \Rightarrow \tau$ and we work in λ-calculus, then the first axiom in Example 2 has a proper subterm of this type, which means that $\mathsf{rhs}_{\overline{x},\tau}(t, \sigma \Rightarrow \tau) = \{\mathsf{app}(t, \mathtt{N}[\overline{x}])\}$ where $\mathtt{N} : [\alpha_1, \dots, \alpha_n]\sigma$ is a fresh metavariable.

The set of generalised Huet-style projection bindings for metavariable $\mathtt{M} : [\alpha_1, \dots, \alpha_n]\beta$ at position i is defined as

$$\{[\mathtt{M}[\overline{x}] \mapsto s] \mid s \in \mathsf{rhs}_{\overline{x},\beta}(x_i, \alpha_i)\}$$

Note that such sets of binding can be precomputed for a given set of types T and axioms E. Once computed, it is possible to use Huet-style bindings to cut down the search space for the E-unification procedure. In particular, the **(project)**, **(imitate)**, and **(iterate)** transition rules may be improved.

Intutively, the improvement comes from the fact that we perform a substitution that brings us as much as possible to the application of the **(mutate)** rule, instead of guessing which path will work.

4 Conclusion

We have described an approach to generalise Huet-style bindings and sketched a way those can be used to reduce the search space for the E-unification procedure for second-order syntax, making a step towards an efficient E-unification procedure. It remains to properly update the procedure, formalising conditions when Huet-style projections must be utilised (we expect it to be similar to full HOU procedure by Vukmirovic, Bentkamp, and Nummelin [13]). It would also be good to see the theoretical performance improvement confirmed with an implementation and benchmarking, especially in comparison with the existing implementations of HOU not based on SOAS.

References

[1] Dominic Duggan. Unification with extended patterns. *Theor. Comput. Sci.*, 206(1-2):1–50, 1998.

[2] Marcelo Fiore and Dmitrij Szamozvancev. Formal Metatheory of Second-Order Abstract Syntax. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[3] 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.

[4] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general E-unification. *Theor. Comput. Sci.*, 67(2&3):203–260, 1989.

[5] Makoto Hamana. Theory and practice of second-order rewriting: Foundation, evolution, and SOL. In Keisuke Nakano and Konstantinos Sagonas, editors, *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings*, volume 12073 of *Lecture Notes in Computer Science*, pages 3–9. Springer, 2020.

[6] Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

[7] D. C. Jensen and Tomasz Pietrzykowski. Mechanizing ω-order type theory through unification. *Theor. Comput. Sci.*, 3(2):123–171, 1976.

[8] Nikolai Kudasov. Functional Pearl: Dependent type inference via free higher-order unification, 2022.

[9] Nikolai Kudasov. Higher-order unification from e-unification with second-order equations and parametrised metavariables. In *UNIF 2022 Worskhop Proceedings. The 36th International Workshop on Unification*. EasyChair, 2022.

[10] Nikolai Kudasov. E-unification for second-order abstract syntax. In Marco Gaboardi and Femke van Raamsdonk, editors, *8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023), FSCD 2023, July 3-July 6, 2023, Rome, Italy*, volume 260 of *LIPIcs*, pages 6:1–6:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[11] Tomer Libal and Dale Miller. Functions-as-Constructors Higher-Order Unification. In Delia Kesner and Brigitte Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, volume 52 of *LIPIcs*, pages 26:1–26:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

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

[13] Petar Vukmirovic, Alexander Bentkamp, and Visa Nummelin. Efficient Full Higher-Order Unification. *Log. Methods Comput. Sci.*, 17(4), 2021.