

Generalising Huet-style Projections for Second-Order Abstract Syntax

Nikolai Kudasov

37th International Workshop on Unification — UNIF (FSCD 2023)

Sapienza Università di Roma, July 2nd, 2023

Lab of Programming Languages and Compilers



Higher-order unification

Higher-order unification (HOU) is a process of solving symbolic equations with functional variables. Consider the following equation that we want to solve for M :

$$\forall g, a. \quad M \ g \ (\lambda z.z \ a) \stackrel{?}{=} g \ a \quad (1)$$

A solution to this problem (called a unifier) is the substitution $\theta = [M \mapsto \lambda x.\lambda y.y \ x]$. Indeed, applying θ to the equation we get β -equivalent terms on both sides:

$$\theta(M \ g \ (\lambda z.z \ a)) = (\lambda x.\lambda y.y \ x) \ g \ (\lambda z.z \ a) \equiv_{\beta} (\lambda y.y \ g) \ (\lambda z.z \ a) \equiv_{\beta} (\lambda z.z \ a) \ g \equiv_{\beta} g \ a = \theta(g \ a)$$

Abstract syntax with binders

Traditionally, HOU algorithms consider only one binder (λ), which is commonly justified by appealing to Higher-Order Abstract Syntax (HOAS):

It is well-known that λ -abstraction is general enough to represent quantification in formulae, abstraction in functional programs, and many other variable-binding constructs (Pfenning and Elliott 1988). (Nipkow and Prehofer 1998, Section 1)

HOAS has received some criticism from both programming language implementors and formalisation researchers, who argue that HOAS and its variants have some practical issues (Cockx 2021; Kmett 2015), such as being hard to work under binders, having issues with general recursion (Kmett 2008), and lacking a formal foundation (Fiore and Szamozvancev 2022).

Second-Order Abstract Syntax

Second-Order Equational Logic (Fiore and Hur 2010) builds upon Second-Order Abstract Syntax (SOAS), an abstract syntax with arbitrary binders and parametrised metavariables.

E -unification for SOAS (Kudasov 2023) provides a sound and complete unification procedure for SOAS modulo a set of second-order equations.

$\beta\eta$ -equivalences of λ -calculus can be presented in SOAS, so E -unification for SOAS generalises HOU.

However, not all transition rules and bindings commonly used in HOU can be transferred to SOAS, since we do not assume syntax of λ -calculus anymore. In particular, Huet-style bindings (a.k.a. partial bindings):

$$M \mapsto \lambda \bar{x}_n. x_i (M_1 \bar{x}_n) \dots (M_1 \bar{x}_n)$$

Second-order signature

Definition 1 (Fiore and Hur 2010, Section 2)

A **second-order signature** $\Sigma = (T, O, | - |)$ is specified by a set of types T , a set of operators¹ O , and an *arity*² function $| - | : O \rightarrow (T^* \times T)^* \times T$.

For an operator $F \in O$, we write $F : (\overline{\sigma}_1.\tau_1, \dots, \overline{\sigma}_n.\tau_n) \rightarrow \tau$ when $|F| = ((\overline{\sigma}_1, \tau_1), \dots, (\overline{\sigma}_n, \tau_n), \tau)$. Intuitively, an operator F takes n arguments each of which binds $n_i = |\sigma_i|$ variables of types $\sigma_{i,1}, \dots, \sigma_{i,n_i}$ in a term of type τ_i .

Example 2

For the untyped λ -calculus, signature $\Sigma = (\{\star\}, \{\text{abs}, \text{app}\}, | - |)$:

$$\text{abs} : \star.\star \rightarrow \star \qquad \text{app} : (\star, \star) \rightarrow \star$$

¹In literature on E -unification, authors use the term *functional symbol* instead.

²Or, perhaps more suitably, a *type signature*.

Signature for simply typed λ -calculus

Example 3

For simply typed λ -calculus with pairs and let-bindings over a set of base types B , signature consists of

1. a set of types $B^{\Rightarrow, \times}$;
2. a type-indexed family of operators:

$$\text{abs}^{\sigma, \tau} : \sigma.\tau \rightarrow \sigma \Rightarrow \tau$$

$$\text{app}^{\sigma, \tau} : (\sigma \Rightarrow \tau, \sigma) \rightarrow \tau$$

$$\text{pair}^{\sigma, \tau} : (\sigma, \tau) \rightarrow \sigma \times \tau$$

$$\text{fst}^{\sigma, \tau} : \sigma \times \tau \rightarrow \sigma$$

$$\text{snd}^{\sigma, \tau} : \sigma \times \tau \rightarrow \tau$$

$$\text{let}^{\sigma, \tau} : (\sigma, \sigma.\tau) \rightarrow \tau$$

Second-order context

In a second-order syntax, we have two types of variables: regular variables and metavariables. Here, metavariables are internalized — they are part of syntax!

Definition 4 (Fiore and Hur 2010, Section 2)

A **typing context** $\Theta \mid \Gamma$ consists of metavariable typings Θ and variable typings Γ .

Metavariable typings are parametrised types: a metavariable of type $[\sigma_1, \dots, \sigma_n]\tau$, when parametrised by terms of type $\sigma_1, \dots, \sigma_n$, will yield a term of type τ .

Example 5

Here is a sample context with metavariable M and variables x, y :

$$M : [\sigma, \sigma \Rightarrow \tau]\tau \mid x : \sigma \Rightarrow \tau, y : \sigma$$

Terms generated by second-order signature

Definition 6 (Fiore and Hur 2010, Section 2)

A judgement for typed **terms** in context $\Theta \mid \Gamma \vdash - : \tau$ is defined by the following rules:

$$\frac{x : \tau \in \Gamma}{\Theta \mid \Gamma \vdash x : \tau} \text{ variables}$$

$$\frac{\begin{array}{l} M : [\sigma_1, \dots, \sigma_n] \tau \in \Theta \\ \text{for all } i = 1, \dots, n \quad \Theta \mid \Gamma \vdash t_i : \sigma_i \end{array}}{\Theta \mid \Gamma \vdash M[t_1, \dots, t_n] : \tau} \text{ metavariables}$$

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

Equational presentation for simply typed λ -calculus

Example 7

The equational presentation of the simply typed λ -calculus with pairs and let-bindings:

$$M : [\sigma]\tau, N : []\sigma \mid \cdot \vdash \text{app}(\text{abs}(x.M[x]), N[]) \equiv M[N[]] : \tau \quad (\beta_\lambda)$$

$$M : []\sigma, N : []\tau \mid \cdot \vdash \text{fst}(\text{pair}(M[], N[])) \equiv M[] : \sigma \quad (\beta_{\pi_1})$$

$$M : []\sigma, N : []\tau \mid \cdot \vdash \text{snd}(\text{pair}(M[], N[])) \equiv N[] : \tau \quad (\beta_{\pi_2})$$

$$M : [\sigma]\tau, N : []\sigma \mid \cdot \vdash \text{let}(N, x.M[x]) \equiv M[N[]] : \tau \quad (\beta_{\text{let}})$$

$$M : []\sigma \Rightarrow \tau \mid \cdot \vdash \text{abs}(x.\text{app}(M[], x)) \equiv M[] : \sigma \Rightarrow \tau \quad (\eta_\lambda)$$

$$M : []\sigma \times \tau \mid \cdot \vdash \text{pair}(\text{fst}(M[]), \text{snd}(M[])) \equiv M[] : \sigma \times \tau \quad (\eta_\times)$$

$$N : []\sigma \mid \cdot \vdash N[] \equiv \text{let}(N[], x.x) \quad (\eta_{\text{let}})$$

$$M : [\sigma]\tau, N : []\sigma \mid \cdot \vdash \text{let}(\text{let}(X[], x.Y[x]), y.M[y]) \equiv \text{let}(X[], x.\text{let}(Y[x], y.M[y])) : \tau \quad (\zeta_{\text{let}})$$

Definition 8

Given an equational presentation E , an **E -unification problem** $\langle \Theta, S \rangle$ is a finite set S of E -unification constraints in a shared metavariable context Θ . We can present an E -unification problem as a formula of the following form:

$$\exists (M_1 : [\overline{\sigma_1}] \tau_1, \dots, M_n : [\overline{\sigma_n}] \tau_n). (\forall (\overline{z_1} : \overline{\rho_1}). s_1 \stackrel{?}{=} t_1 : \tau_1) \wedge \dots \wedge (\forall (\overline{z_k} : \overline{\rho_k}). s_k \stackrel{?}{=} t_k : \tau_k)$$

Example 9

An example of E -unification problem for simply typed λ -terms:

$$\exists (M : [\sigma \Rightarrow \tau, (\sigma \Rightarrow \tau) \Rightarrow \tau] \tau).$$

$$\forall (g : \sigma \Rightarrow \tau, y : \sigma).$$

$$M[g, \text{abs}(x. \text{app}(x, y))] \stackrel{?}{=} \text{app}(g, y) : \tau$$

Definition 10

Given an E -unification problem $\langle \Theta, S \rangle$, a metavariable substitution $\xi : \Theta \rightarrow \Xi$ is called an **E -unifier** for $\langle \Theta, S \rangle$ if for all constraints $(\Theta \mid \Gamma_{\forall} \vdash s \stackrel{?}{=} t : \tau) \in S$ we have

$$\Xi \mid \Gamma_{\forall} \vdash \xi s \equiv_E \xi t : \tau$$

We write $U_E(S)$ for the set of all E -unifiers for $\langle \Theta, S \rangle$.

Example 11

Consider a unification problem $\langle \Theta, S \rangle$ for the simply typed λ -calculus:

$$\Theta = M : [\sigma \Rightarrow \tau, (\sigma \Rightarrow \tau) \Rightarrow \tau] \tau$$

$$S = \{ \Theta \mid g : \sigma \Rightarrow \tau, y : \sigma \vdash M[g, \text{abs}(x.\text{app}(x, y))] \stackrel{?}{=} \text{app}(g, y) : \tau \}$$

Substitution $\xi = [M[z_1, z_2] \mapsto \text{app}(z_2, z_1)] : \Theta \rightarrow \cdot$ is an E -unifier for $\langle \Theta, S \rangle$.

Unification Procedure Overview

Overall structure is straightforward. Importantly, we aim for completeness, but not efficiency.

1. We classify kinds of partial solutions.
2. We define simple transition rules.
3. We determine when to stop (succeed or fail).
4. The procedure then repeatedly applies rules non-deterministically, until stop.

For the equational part, we took inspiration from the complete sets of transformations for general (first-order) E -unification (Gallier and Snyder 1989). For unification of metavariables, we took inspiration primarily from (Jensen and Pietrzykowski 1976).

JP-style projection for M . If $M : [\sigma_1, \dots, \sigma_k] \tau$ and $\sigma_i = \tau$ then

$\zeta = [M[\bar{z}] \mapsto z_i]$ is a JP-style projection binding

Imitation for M . If $M : [\sigma_1, \dots, \sigma_k] \tau$, $F : (\bar{\alpha}_1.\beta_1, \dots, \bar{\alpha}_n.\beta_n) \rightarrow \tau$ and

$M_i : [\sigma_1, \dots, \sigma_k, \bar{\alpha}_i] \beta_i$ for all i ,

$\zeta = [M[\bar{z}] \mapsto F(\bar{x}_1.M_1[\bar{z}, \bar{x}_1], \dots, \bar{x}_n.M_n[\bar{z}, \bar{x}_n])]$ is an imitation binding

Elimination for M . If $M : [\sigma_1, \dots, \sigma_k] \tau$ and $1 \leq j_1 < j_2 < \dots < j_{n-1} < j_n \leq k$ such that $E : [\sigma_{j_1}, \dots, \sigma_{j_n}] \tau$ then

$\zeta = [M[\bar{z}] \mapsto E[z_{j_1}, \dots, z_{j_n}]]$ is a (parameter) elimination binding

Identification of M and N. If $M : [\sigma_1, \dots, \sigma_k]\tau$, $N : [\nu_1, \dots, \nu_l]\tau$,

$I : [\sigma_1, \dots, \sigma_k, \nu_1, \dots, \nu_l]\tau$,

$M_i : [\sigma_1, \dots, \sigma_k]\nu_i$ for all $i \in \{1, \dots, l\}$, and $N_j : [\nu_1, \dots, \nu_l]\sigma_j$ for all $j \in \{1, \dots, k\}$ then

$\zeta = [M[\bar{z}] \mapsto I[\bar{z}, M_1[\bar{z}], \dots, M_l[\bar{z}]], N[\bar{y}] \mapsto I[N_1[\bar{y}], \dots, N_k[\bar{y}], \bar{y}]]$ is an identification binding

Iteration for M. If $M : [\sigma_1, \dots, \sigma_k]\tau$, $F : (\overline{\alpha_1}.\beta_1, \dots, \overline{\alpha_n}.\beta_n) \rightarrow \gamma$, $H : [\sigma_1, \dots, \sigma_k, \gamma]\tau$, and $M_i : [\sigma_1, \dots, \sigma_k, \overline{\alpha_i}]\beta_i$ for all i , then

$\zeta = [M[\bar{z}] \mapsto H[\bar{z}, F(\overline{x_1}.M_1[\bar{z}, \overline{x_1}], \dots, \overline{x_n}.M_n[\bar{z}, \overline{x_n}])]]$ is an iteration binding

Transition Rules (delete)

Definition 12 (delete)

If a constraint has the same term on both sides, we can **delete** it:

$$(\Theta \mid \Gamma_{\forall} \vdash t \stackrel{?}{=} t : \tau) \xrightarrow{\text{id}} \emptyset$$

Transition Rules (decompose)

Definition 13 (decompose)

We define two variants of this rule:

1. Let $F : (\overline{\sigma}_1.\tau_1, \dots, \overline{\sigma}_n.\tau_n) \rightarrow \tau$, then we can **decompose** a constraint with F on both sides into a set of constraints for each pair of (scoped) subterms:

$$(\Theta \mid \Gamma_{\forall} \vdash F(\overline{x}.t) \stackrel{?}{=} F(\overline{x}.s) : \tau) \xrightarrow{\text{id}} \{\Theta \mid \Gamma_{\forall}, \overline{x}_i : \overline{\sigma}_i \vdash t \stackrel{?}{=} s : \tau_i\}_{i \in \{1, \dots, n\}}$$

2. Let $M : [\sigma_1, \dots, \sigma_n]\tau$, then we can **decompose** a constraint with M on both sides into a set of constraints for each pair of parameters:

$$(\Theta \mid \Gamma_{\forall} \vdash M[\overline{t}] \stackrel{?}{=} M[\overline{s}] : \tau) \xrightarrow{\text{id}} \{\Theta \mid \Gamma_{\forall} \vdash t \stackrel{?}{=} s : \sigma_i\}_{i \in \{1, \dots, n\}}$$

Transition Rules (imitate)

Definition 14 (imitate)

For flex-rigid constraints with a metavariable $M : [\overline{\sigma}_s]\tau$ and an operator

$F : (\overline{\sigma}_1.\tau_1, \dots, \overline{\sigma}_n.\tau_n) \rightarrow \tau$ we can **imitate** the rigid side using an imitation binding:

$$(\Theta \mid \Gamma_{\forall} \vdash M[\overline{s}] \stackrel{?}{=} F(\overline{x}.t) : \tau) \xrightarrow{[M[\overline{z}_s] \mapsto F(\overline{x}.T[\overline{z}_s, \overline{x}])]} \{\Theta \mid \Gamma_{\forall} \vdash F(\overline{x}.T[\overline{s}, \overline{x}]) \stackrel{?}{=} F(\overline{x}.t) : \tau\}$$

Note that **(imitate)** can be followed up by an application of **(decompose)** rule.

Definition 15 (project)

For constraints with a metavariable $M : [\overline{\sigma}_s]\tau$ and a term $u : \tau$ we can produce a JP-style projection binding:

$$(\Theta \mid \Gamma_{\forall} \vdash M[\overline{s}] \stackrel{?}{=} u : \tau) \xrightarrow{[M[\overline{z}] \mapsto z_i]} \{\Theta \mid \Gamma_{\forall} \vdash s_i \stackrel{?}{=} u : \tau\}$$

Transition Rules (mutate)

Definition 16 (mutate)

For constraints where one of the sides matches an axiom in E :

$$\Xi \mid \cdot \vdash l \equiv r : \tau$$

We rewrite the corresponding side (here, ξ instantiates the axiom in context $\Theta \mid \Gamma_{\forall}$).

$$(\Theta \mid \Gamma_{\forall} \vdash t \stackrel{?}{=} s : \tau) \xrightarrow{\text{id}} \{\Theta \mid \Gamma_{\forall} \vdash t \stackrel{?}{=} \xi l : \tau\} \uplus \{\Theta \mid \Gamma_{\forall} \vdash \xi r \stackrel{?}{=} s : \tau\}$$

Transition Rules (identify)

Definition 17 (identify)

When a constraint consists of a pair of distinct metavariables $M : [\sigma_1, \dots, \sigma_k]\tau$ and $N : [\gamma_1, \dots, \gamma_l]\tau$, we can use an identification binding:

$$(\Theta \mid \Gamma_{\forall} \vdash M[\bar{s}] \stackrel{?}{=} N[\bar{t}]) \xrightarrow{[M[\bar{z}] \mapsto I[\bar{z}, \overline{M'[\bar{z}}]], N[\bar{y}] \mapsto I[\overline{N'[\bar{y}}], \bar{y}]]} \{\Theta \mid \Gamma_{\forall} \vdash I[\bar{s}, \overline{M'[\bar{s}}]] \stackrel{?}{=} I[\overline{N'[\bar{u}}], \bar{u}]\}$$

Transition Rules (eliminate)

Definition 18 (eliminate)

When a constraint has the same metavariable $M : [\sigma_1, \dots, \sigma_n] \tau$ on both sides and there is a sequence $(j_k)_{k=1}^n$ such that $s_{j_k} = u_{j_k}$ for all $k \in \{1, \dots, n\}$, then we can **eliminate** every other parameter and leave the remaining terms identical:

$$(\Theta \mid \Gamma_{\forall} \vdash M[\bar{s}] \stackrel{?}{=} M[\bar{t}]) \xrightarrow{[M[\bar{z}] \mapsto E[z_{j_1}, \dots, z_{j_n}]]} \emptyset$$

Transition Rules (iterate)

Definition 19 (iterate)

When a constraint consists of a pair of (possibly, identical) metavariables

$M : [\sigma_1, \dots, \sigma_k]\mathcal{T}$ and $N : [\gamma_1, \dots, \gamma_l]\mathcal{T}$, we can use an iteration binding:

$$(\Theta \mid \Gamma_{\forall} \vdash M[\bar{s}] \stackrel{?}{=} N[\bar{t}]) \xrightarrow{[M[\bar{z}] \mapsto H[\bar{z}, F(\overline{x.K[\bar{z}, \bar{x}]})]]} \{\Theta \mid \Gamma_{\forall} \vdash H[\bar{s}, F(\overline{x.K[\bar{z}, \bar{x}]})] \stackrel{?}{=} N[\bar{t}]\}$$

Example for (iterate) rule

The following example demonstrates the importance of iteration by an arbitrary operator to introduce variables into scope:

Example 20

Consider a unification problem for simply-typed λ -calculus:

$$\exists M : [\sigma \Rightarrow \sigma \Rightarrow \tau](\sigma \Rightarrow \tau)$$

$$\forall f : \sigma \Rightarrow \sigma \Rightarrow \sigma \Rightarrow \sigma \Rightarrow \tau.$$

$$M[\lambda x. \lambda y. f x y x x] \stackrel{?}{=} M[\lambda x. \lambda y. f y y x y] : \sigma \Rightarrow \tau$$

It has the following E -unifier: $\zeta = [M[g] \mapsto \lambda z. g z z]$. To construct this unifier from bindings, we start with iteration binding $[M[g] \mapsto I[g, \lambda z. M_1[g, z]]]$, introducing the lambda abstraction, which is followed by a projection $[I[g, r] \mapsto r]$, which is followed by another iteration (to introduce application), and so on.

Unification Procedure

Definition 21

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 $M[\dots] \stackrel{?}{=} F(\dots)$, apply (**imitate**) or (**project**) rules (non-det.).
6. If there is a constraint $M[\dots] \stackrel{?}{=} x$, 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**).

Huet-style projection (HOU)

Definition 22

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

$$M \mapsto \lambda \bar{x}. x_i (\mathbf{H}_1 \bar{x}) \dots (\mathbf{H}_m \bar{x})$$

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

Huet-style projection (HOU)

1. The type of i -th parameter ($\alpha_i = \gamma_1 \Rightarrow \dots \Rightarrow \gamma_m \Rightarrow \beta$) determines if this parameter can be used to extract the result of type β .
2. The type of 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 -abstraction.
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.

Huet-style projection (pairs)

Consider the constraint

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

Assuming $f: \sigma \Rightarrow \tau$ and $y: \sigma$, the type of the only parameter to M is $((\sigma \Rightarrow \tau) \Rightarrow \tau) \times (\sigma \Rightarrow \tau)$. It is clear that we should be able to 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:

$$M \mapsto \lambda x_1. \pi_1 \ x_1 \ (H_1 \ x_1)$$

We observe the following from this example:

1. The type of a parameter to 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 .

Huet-style projection (pairs)

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

The set of possible Huet-style projection bindings for metavariable

$\mathbf{M} : \alpha_1 \Rightarrow \dots \alpha_n \Rightarrow \beta$ at position i is defined as

$$\{[\mathbf{M} \mapsto \lambda \bar{x}.s \mid s \in \text{rhs}_{\bar{x},\beta}(x_i, \alpha_i)]\}$$

Generalised Huet-style projection

Assume that $M : [\alpha_1, \dots, \alpha_n]\beta$. Let $x : \alpha_1, \dots, x_n : \alpha_n$. We define the set of right-hand side formulas $\text{rhs}_{\bar{x},\beta}$ recursively as follows:

1. $\text{rhs}_{\bar{x},\beta}(t, \beta) = \{t\}$
2. $\text{rhs}_{\bar{x},\beta}(t, \delta) = \bigcup_{\langle l, \xi \rangle \in L} \text{rhs}_{\bar{x},\beta}([y \mapsto t]\xi l, \varepsilon)$

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. $\text{rhs}_{\bar{x},\beta}(t, \delta) = \emptyset$ otherwise

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

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



Results and Future work




1. An (untyped) version of Huet-style projection has been implemented as a heuristic for (dependent) type inference (Kudasov 2022).
2. A sketch of the typed version of generalised Huet-style bindings.




Unresolved questions and future work:



1. Some efficient HOU algorithms (Vukmirovic, Bentkamp, and Nummelin 2021) rely on Huet-style bindings for flex-rigid constraints, to avoid infinitely branching (**iterate**) rule. However, E -unification for SOAS already avoids that and only applies (**iterate**) only for flex-flex pairs. Thus, it is unclear how much search space is reduced with Huet-style bindings (although it definitely helps).
2. The version of (**iterate**) rule for SOAS uses an arbitrary operator but in HOU Huet-style binding is used. Can do the same for SOAS preserving completeness?
3. Completeness proof has to be updated.



Thank you!

-  Cockx, Jesper (Nov. 2021). *1001 Representations of Syntax with Binding*. <https://jesper.sikanda.be/posts/1001-syntax-representations.html>. Accessed: 2023-01-21.
-  Fiore, Marcelo and Chung-Kil Hur (2010). “Second-Order Equational Logic (Extended Abstract)”. In: *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*. Ed. by Anuj Dawar and Helmut Veith. Vol. 6247. Lecture Notes in Computer Science. Springer, pp. 320–335. DOI: [10.1007/978-3-642-15205-4_26](https://doi.org/10.1007/978-3-642-15205-4_26). URL: https://doi.org/10.1007/978-3-642-15205-4%5C_26.

-  Fiore, Marcelo and Dmitriy Szamozvancev (Jan. 2022). “Formal Metatheory of Second-Order Abstract Syntax”. In: *Proc. ACM Program. Lang.* 6.POPL. DOI: [10.1145/3498715](https://doi.org/10.1145/3498715). URL: <https://doi.org/10.1145/3498715>.
-  Gallier, Jean H. and Wayne Snyder (1989). “Complete Sets of Transformations for General E-Unification”. In: *Theor. Comput. Sci.* 67.2&3, pp. 203–260. DOI: [10.1016/0304-3975\(89\)90004-2](https://doi.org/10.1016/0304-3975(89)90004-2). URL: [https://doi.org/10.1016/0304-3975\(89\)90004-2](https://doi.org/10.1016/0304-3975(89)90004-2).
-  Jensen, D. C. and Tomasz Pietrzykowski (1976). “Mechanizing ω -Order Type Theory Through Unification”. In: *Theor. Comput. Sci.* 3.2, pp. 123–171. DOI: [10.1016/0304-3975\(76\)90021-9](https://doi.org/10.1016/0304-3975(76)90021-9). URL: [https://doi.org/10.1016/0304-3975\(76\)90021-9](https://doi.org/10.1016/0304-3975(76)90021-9).

-  Kmett, Edward (Mar. 2008). *Rotten Bananas*.
<http://comonad.com/reader/2008/rotten-bananas/>. Accessed: 2023-01-21.
-  — (Dec. 2015). *Bound*. <https://www.schoolofhaskell.com/user/edwardk/bound>.
Accessed: 2023-01-21.
-  Kudasov, Nikolai (2022). *Functional Pearl: Dependent type inference via free higher-order unification*. arXiv: [2204.05653](https://arxiv.org/abs/2204.05653) [cs.LG].

-  Kudasov, Nikolai (2023). “E-Unification for Second-Order Abstract Syntax”. In: *8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023)*. Ed. by Marco Gaboardi and Femke van Raamsdonk. Vol. 260. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 10:1–10:22. ISBN: 978-3-95977-277-8. DOI: [10.4230/LIPIcs.FSCD.2023.10](https://doi.org/10.4230/LIPIcs.FSCD.2023.10). URL: <https://drops.dagstuhl.de/opus/volltexte/2023/17994>.
-  Nipkow, Tobias and Christian Prehofer (1998). “Higher-Order Rewriting and Equational Reasoning”. In: *Automated Deduction — A Basis for Applications. Volume I: Foundations*. Ed. by W. Bibel and P. Schmitt. Vol. 8. Applied Logic Series. Kluwer, pp. 399–430.

-  Pfenning, Frank and Conal Elliott (1988). “Higher-Order Abstract Syntax”. In: *Proceedings of the ACM SIGPLAN’88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. Ed. by Richard L. Wexelblat. ACM, pp. 199–208. DOI: [10.1145/53990.54010](https://doi.org/10.1145/53990.54010). URL: <https://doi.org/10.1145/53990.54010>.
-  Vukmirovic, Petar, Alexander Bentkamp, and Visa Nummelin (2021). “Efficient Full Higher-Order Unification”. In: *Log. Methods Comput. Sci.* 17.4. DOI: [10.46298/lmcs-17\(4:18\)2021](https://doi.org/10.46298/lmcs-17(4:18)2021). URL: [https://doi.org/10.46298/lmcs-17\(4:18\)2021](https://doi.org/10.46298/lmcs-17(4:18)2021).