Understanding Higher Order Unification via Explicit Substitutions and Patterns

Flávio L. C. de Moura[1⋆]
Departamento de Matemática, Universidade de Brasília, Brasília D.F., Brasil.
flavio@mat.unb.br

1 Introduction

Unification plays an essential role in many areas of computation such as automated deduction, programming languages, proof assistants, etc. First order unification is decidable [Rob65] and the solutions are unique. In fact, every unifiable set has a *most general unifier (mgu)*. Applications of first order unification include implementations of type inference algorithms and of the resolution principle used in programming languages, such as Prolog. For higher order languages, the unification problem is undecidable [Luc72,Hue73], even for the second order case[Gol81], and the notion of mgu no more exists. The importance of first order unification notwithstanding, first order languages have limited expressive power which raise difficulties for manipulation of higher order features, such as higher order functions, that is, functions that take another functions as argument or return another functions as result. The (untyped) λ-calculus is a higher order language, with a simple and compact syntax, adequate to express computable functions, and hence higher order functions. Unification in the λ-calculus, or simply *Higher Order Unification (HOU)*, is more complex than first order unification because it involves semantical aspects of the language.

In [DHK00], an algorithm for unification in the $\lambda\sigma$-calculus of explicit substitutions [ACCL91] is presented. This algorithm is a generalisation of the Huet's algorithm, although no formal comparison between them is provided. In this work, we formally compare these two methods proving that using particular strategies of the $\lambda\sigma$-HOU algorithm we can simulate the algorithm of Huet. Afterwards, we introduce the decidable subclass of λ-terms known as higher order patterns as well as some of its properties.

2 Higher Order Unification via Explicit Substitutions

The λ-calculus is based on a notion of substitution that belongs to a meta-language. Such a notion is necessary because the substitution process adopts renaming of bound variables in order to avoid variable capture. A natural solution for this drawback is to turn the substitution process explicit. The first mechanism that made the substitutions explicit was the $\lambda\sigma$-calculus [ACCL91]. The $\lambda\sigma$-terms are built over two different sets of entities: $\texttt{terms}\ \ t ::= \underline{1} \mid X \mid (t\ t) \mid (\lambda.t) \mid t[s]$ and $\texttt{substitutions}\ \ s ::= id \mid \uparrow \mid t \cdot s \mid s \circ s$, where constants and bound variables are coded as de Bruijn indexes and (free variables) $X, Y, Z, \ldots$ range over the set $\mathcal{X}$ of the meta-variables. In this calculus, when a substitution s is applied to a term t we internalise this as $t[s]$. Simultaneous substitutions are represented as lists of terms with the usual operator *cons* (written as "."), an operator for the empty list (written id which represents the identity substitution) and the operator $\uparrow$ which represents the infinite substitution $\underline{2}.\underline{3}.\ldots$.

In [DHK00], a generalisation of the Huet's unification method, based on the $\lambda\sigma$-style of explicit substitutions, is presented. Nevertheless, no formal comparison among these methods is provided. Here, we compare these two methods proving that the SIMPL and MATCH procedures of Huet can be simulated in the $\lambda\sigma$-calculus for both β- and $\beta\eta$-unification.

A unification problem in the $\lambda\sigma$-calculus is written as a disjunction of existentially quantified conjunctions of the form $\bigvee_{j\in J} \exists \overrightarrow{w}_j \bigwedge_{i\in I_j} s_i =^?_{\lambda\sigma} t_i$ where s_i and t_i are $\lambda\sigma$-terms of the same type. If $|J| = 1$ then the unification problem is called a *unification system*. The process of translating a unification problem P from the simply typed λ-calculus to the correspondent unification problem P_F in the typed $\lambda\sigma$-calculus is done by the *precooking* translation (cf. [DHK00]).

The SIMPL$_{\lambda\sigma}$ procedure takes a unification system with at least one rigid-rigid equation as argument according to the description given below.

The inference rules used to manipulate rigid-rigid equations are given in the Table 1. Note that in the rule **Dec-λ** the binder λ_A can be removed because bound and free variables belong to different sets and can always be distinguished.

⋆ Author supported by the Brazilian CAPES Foundation (PhD studentship 2002-2006 and PhD sandwich studentship 2004-2005 at Heriot-Watt University - Scotland - UK).

In the $\lambda\sigma$-calculus with **Eta**-conversion, the second step (**Dec-App-**λ) in the below description may be eliminated because it never applies since **Eta**-long terms of functional types are always abstractions and, if they have the same type, they also have the same number of external abstractors.

Procedure SIMPL$_{\lambda\sigma}$

INPUT: A unification system P with at least one rigid-rigid equation eq.

WHILE there exists a rigid-rigid equation in P DO

1. Apply **Dec-**λ as much as possible to eq, generating the new equation eq'.
2. Apply **Dec-App-**λ to eq', call $\overline{P}$ the resulting unification system and return $\overline{P}$.
3. Apply **Dec-Fail** or **Dec-App** to eq', call $\overline{P}$ the resulting unification system after an application of the rules **Replace** and **Normalise** and then return $\overline{P}$.

DONE.

If $\overline{P}$ contains a flexible-rigid equation then return $\overline{P}$, else stop returning a success status.

OUTPUT: A success status or an equivalent unification system $\overline{P}$ without rigid-rigid equations.

Dec-λ	$\dfrac{P \wedge \lambda_A e_1 =^? \lambda_A e_2}{P \wedge e_1 =^? e_2}$	**Dec-App**	$\dfrac{P \wedge \underline{\mathtt{n}}(e_1^1, \ldots, e_p^1) =^? \underline{\mathtt{n}}(e_1^2, \ldots, e_p^2)}{P \wedge e_1^1 =^? e_1^2 \wedge \ldots \wedge e_p^1 =^? e_p^2}$
Dec-Fail	$\dfrac{P \wedge \underline{\mathtt{n}}(e_1^1, \ldots, e_{p_1}^1) =^? \underline{\mathtt{m}}(e_1^2, \ldots, e_{p_2}^2)}{P \wedge \bot}$, if $\mathtt{m} \neq \mathtt{n}$.	**Dec-App-**λ	$\dfrac{P \wedge \lambda_A e =^? \underline{\mathtt{m}}(e_1^2, \ldots, e_{p_2}^2)}{P \wedge \bot}$

Table 1. Unification Rules for the $\lambda\sigma$-calculus

Proposition 1. *The application of the procedure SIMPL$_{\lambda\sigma}$ to the unification problem P terminates and, if the result is not a terminal status, it produces an equivalent unification problem $\overline{P}$ without rigid-rigid equations, in the $\lambda\sigma$-calculus with or without the* **Eta***-conversion.*

Proposition 2. *There is a correspondence between the procedures SIMPL$_{\lambda\sigma}$ and SIMPL such that if P is a unification system in the simply typed λ-calculus and P_F is its precooked image in the typed $\lambda\sigma$-calculus, then:*

- *SIMPL(P) fails $\Leftrightarrow$ SIMPL$_{\lambda\sigma}(P_F)$ contains the constant $\bot$;*
- *SIMPL(P) stops reporting a success status $\Leftrightarrow$ SIMPL$_{\lambda\sigma}(P_F)$ stops returning a success status;*
- *SIMPL(P) returns $\overline{P}$ $\Leftrightarrow$ SIMPL$_{\lambda\sigma}(P_F)$ returns $\overline{P}_F$.*

The MATCH$_{\lambda\sigma}$ procedure takes a unification system as argument and return an equivalent unification problem. In the unification method for the $\lambda\sigma$-calculus, graftings are build from solved equations which justifies our construction of the procedure MATCH$_{\lambda\sigma}$. The other rules used in the unification algorithm are in Table 2. The subscripts indicate the position of the unification system in the matching tree.

Procedure MATCH$_{\lambda\sigma}$ (with Eta)

INPUT: A unification system P_q with at least one flexible-rigid equation eq.

1. Apply **Dec-**λ as much as possible to eq and call eq' the resulting equation.
2. Apply the strategy **Exp-**λ, **Replace** and **Normalise** as much as possible to $(P \setminus eq) \wedge eq'$ and call $\overline{P}_q$ the resulting unification system.
3. Apply **Exp-App** to $\overline{P}_q$ and let $P_{q1} \vee \ldots \vee P_{qk}$ be the resulting unification problem.

OUTPUT: A disjunction of the form $P_{q1} \vee \ldots \vee P_{qk}$.

In the $\lambda\sigma$-calculus, our version of the Huet's algorithm can be seen as successive calls to the procedures SIMPL$_{\lambda\sigma}$ and MATCH$_{\lambda\sigma}$ according to the following description. For the sake of clarity, we will denote the disjunction $P_{q1} \vee \ldots \vee P_{qk}$ by S_q, and $S_\epsilon = P_\epsilon$.

Main Procedure

INPUT: A unification problem S_ϵ.

FOR each P_j not in $\lambda\sigma$-solved form in S_k DO:

1. If P_j contains a rigid-rigid equation then apply SIMPL$_{\lambda\sigma}$ to it, else rename P_j to $\overline{P}_j$ and go to next step. If SIMPL$_{\lambda\sigma}$ does not return a success status then let $\overline{P}_j$ be the resulting unification system and go to next step.

2. If $\overline{P}_j$ contains the constant $\perp$ then stop reporting a failure status; else, if $\overline{P}_j$ contains a flexible-rigid equation then go to next step, else stop reporting a success status.
3. Apply MATCH$_{\lambda\sigma}$ to $\overline{P}_j$, call $S_j := P_{j1} \vee \ldots \vee P_{jr}$ the new unification problem and go to next step.
4. Apply **Replace** and then **Normalise** to each P_{js} in S_j and call $\overline{S}_j$ the resulting unification problem.

DONE.

If $\overline{S}_j$ contains a unification system which is not in $\lambda\sigma$-solved form then apply the above FOR to it, else stop reporting a success status.

OUTPUT: A success or a failure status and in the former case the solutions are the solved equations whose left hand side are the meta-variables of the initial problem. If the initial problem is non-unifiable the algorithm may not terminate.

Lemma 1. *Let* $X[a_1.\ldots.a_p.\uparrow^n](e_1,\ldots,e_k) =^?_{\lambda\sigma} t$ *be a flexible-rigid equation in the* $\lambda\sigma$*-calculus such that the type of* X *is given by* $A_1 \to \cdots \to A_k \to B$ *(with* $k \geq 0$ *and* B *atomic) and* X *does not occur in* t*. Then the* $\lambda\sigma$*-normal form of this equation, after some atomisation steps generated by successive applications of the strategy* **Exp-**λ*,* **Replace** *and* **Normalise***, produces an equivalent flexible-rigid equation of the form* $X_k[e_k.\cdots.e_1.a_1.a_2.\ldots.a_p.\uparrow^n] =^?_{\lambda\sigma} t \wedge X =^?_{\lambda\sigma} \lambda_{A_1}\ldots\lambda_{A_k}.X_k \wedge Q$ *where* X_k *is a new meta-variable with atomic type* B *and* Q *is a conjunction of solved equations.*

Proposition 3. *Let* $\lambda_{A_1}\ldots\lambda_{A_k}.X[\uparrow^k](e_{1_F},\ldots,e_{k_F}) =^?_{\lambda\sigma} \lambda_{A_1}\ldots\lambda_{A_k}.t_F$ *be a flexible-rigid equation in* **Eta**-*long normal form which is in the image of the precooking translation and,* $X_k[e_{k_F}.\cdots.e_{1_F}.\uparrow^k] =^?_{\lambda\sigma} t_F$ *its equivalent form after some atomisation steps according to the strategy given in Lemma 1. Then for each new equation generated by the rule* **Exp-App** *of the form* $X_k =^?_{\lambda\sigma} \underline{\mathtt{r}}(H_1,\ldots,H_s)$ *with* $\underline{\mathtt{r}} \in R_i \cup R_p$*, there exists a 1-1 correspondence between the solutions in the* $\lambda\sigma$*-calculus and in the pure* λ*-calculus in the following sense: for each element in* R_p *there exists a corresponding projection in the pure* λ*-calculus, and* $R_i \neq \emptyset$ *if and only if there exists an imitation substitution in the pure* λ*-calculus.*

Procedure MATCH$_{\lambda\sigma}$ (without Eta)

INPUT: A unification problem P_q with at least one flexible-rigid equation eq.

1. Apply **Dec-**λ as much as possible to eq and call eq' the resulting equation.
2. Apply **Exp** or **Exp2** to $(P_q \setminus eq) \wedge eq'$ and call $P_{q1} \vee \ldots \vee P_{qk}$ the resulting unification problem.

OUTPUT: A disjunction of the form $P_{q1} \vee \ldots \vee P_{qk}$.

3 Higher Order Patterns

Although the undecidability of the HOU problem, there is at least one important reduct of λ-terms where this problem is decidable. This reduct was discovered by Miller [Mil91] and is known as *(higher order) patterns*. Pattern unification is also unitary and is, in fact, a generalisation of first order unification. The characterisation of such classes of λ-terms are fundamental since it simplifies significantly the unification process allowing better implementations of HOU algorithms. The importance of patterns is justified by the fact that they frequently turn up in practice. In this section, we present an algorithm for pattern unification in the λ-calculus in de Bruijn notation with η-conversion.

A Higher Order Pattern Unification (HOPU) problem is a list of equations of the form $t =^? s$, where t and s are patterns of the same type. The use of list as data structure is important due to termination. For instance, consider the unification problem $\underline{\mathtt{n}}(X) =^? \wedge Y =^? X$. Applying the **Flex-Rig** rule, we get $\underline{\mathtt{n}}(X) =^? \underline{\mathtt{n}}(H_1) \wedge \underline{\mathtt{n}}(H_1) =^? X$ with the substitution $Y/\underline{\mathtt{n}}(H_1)$. After an application of the rule **Dec-App** we get $X =^? H_1 \wedge \underline{\mathtt{n}}(H_1) =^? X$ which is equivalent to the original problem up to renaming of meta-variables, and so applying the same strategy to the second equation of the current problem, we can generate an infinite reduction. The manipulation of rigid-rigid equations are done by the same rules **Dec-**λ, **Dec-App** and **Dec-Fail** of Table 1 where conjunction should be seen as the usual *cons* list operator, written as "::". The rules for flexible-rigid and flexible-flexible equations are given in Table 3 and are based on [Nip93].

Procedure SIMPL$_p$

INPUT: a pattern unification problem P with at least one rigid-rigid equation eq.

1. Apply **Dec-**λ as much as possible to the equation eq and call eq' the resulting equation.
2. Apply **Dec-App** or **Dec-Fail** to the equation eq'. Call P' the resulting unification problem after eliminating all trivial equations.

3. If P' contains a rigid-rigid equation, say eq, then go to step 1 else call $\overline{P}$ the current unification problem and return $\overline{P}$.

OUTPUT: an equivalent pattern unification problem $\overline{P}$ without rigid-rigid equations.

The Main Procedure

INPUT: a pattern unification problem P_1.

1. If P_i has a rigid-rigid equation then apply SIMPL$_p$ to P_i, call $\overline{P_i}$ the resulting equivalent unification problem and go to the next step; else simply rename P_i to $\overline{P_i}$ go to step 2.
2. If $\overline{P_i}$ contains the constant $\perp$ then stop returning a failure status, else if $\overline{P_i}$ is empty then go to step 3; else select a an equation eq in $\overline{P_i}$. If eq is a flexible-rigid equation then apply the **Flex-Rig** rule, else apply **Flex-Flex1** or **Flex-Flex2** and, in either case, call P_{i+1} the resulting unification problem after eliminating all trivial equations and call σ_i the generated substitution.
3. Stop reporting a success status and the substitution solution is given by the composition $\sigma_{i-1}\sigma_{i-2}\ldots\sigma_1$.

OUTPUT: A success or a failure status according to P_1 is unifiable or not, respectively. In the former case, if P_i is empty but P_{i-1} is not, then the most general unifier for P_1 is given by $\sigma_{i-1}\sigma_{i-2}\ldots\sigma_1$. If $i = 1$ then σ_0 denotes the empty substitution.

Another important property of the patterns is that some general unification problems can be embedded into a pattern unification problem and hence this reduct is not very restrictive. For instance, the term $\lambda_A.\underline{\mathtt{n}}(\lambda_B.\underline{\mathtt{m}}(X(\underline{\mathtt{m}}, Y(\underline{\mathtt{k}}))), Y(Z))$ can be flattened to $\lambda_A.\underline{\mathtt{n}}(\lambda_B.\underline{\mathtt{m}}(X_1(\underline{\mathtt{2}}, \underline{\mathtt{1}})), X_2)$ with constraints $X_1 = \lambda_A\lambda_B.X(\underline{\mathtt{m}}, Y(\underline{\mathtt{k}}))$ and $X_2 = Y(Z)$. The possibility of such translation to patterns is important since it simplifies the unification problem. In fact, if the λ-terms t_1 and t_2 can be flattened into patterns (with constraints) t_1' and t_2' then if t_1' and t_2' are not unifiable neither are t_1 nor t_2 (cf. [Pre97]).

4 Conclusions and Future Work

In this work, we compared two methods for higher order unification. Using particular strategies of the inference rules of the HOU method via the $\lambda\sigma$-calculus of explicit substitutions, we build the $\lambda\sigma$-version of the procedures SIMPL and MATCH of Huet, called SIMPL$_{\lambda\sigma}$ and MATCH$_{\lambda\sigma}$, respectively. We proved that there is a correspondence between the procedures SIMPL$_{\lambda\sigma}$ and SIMPL and also between the procedures MATCH$_{\lambda\sigma}$ and MATCH. Our correspondences allow us to formalise the fact that the method of HOU via explicit substitution à la [DHK00] is a generalisation of the HOU method of [Hue75]. In addition, we presented an algorithm for higher order patterns unification in de Bruijn notation with η-conversion.

We believe this formalisation is worthwhile for having a good understanding of the novel alternative of HOU via general explicit substitutions calculi and contributes to current research related to the design and implementation of higher order computational environments.

An important work, still in development, is a complete characterisation of the patterns in the λs_e- and suspension styles of explicit substitutions that will permit better comparisons among these calculi as well as better implementations of HOU algorithms.

References

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *J. of Func. Programming*, 1(4):375–416, 1991.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157:183–235, 2000.

[Gol81] W. Goldfarb. The Undecidability of the Second-Order Unification Problem. *TCS*, 13(2):225–230, 1981.

[Hue73] G. Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267, April 1973.

[Hue75] G. Huet. A Unification Algorithm for Typed λ-Calculus. *TCS*, 1:27–57, 1975.

[Luc72] C. L. Lucchesi. The undecidability of the unification problem for third order languages. Technical report, University of Waterloo, 1972.

[Mil91] D. Miller. A Logic Programming Language with Lambda-abstraction, Function Variables, and Simple Unification. *Logic and Computation*, 1(4):497–536, 1991.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.

[Pre97] C. Prehofer. Progress in Theoretical Computer Science. In R. V. Book, editor, *Solving Higher-Order Equations: From Logic to Programming*. Birkhäuser, 1997.

[Rob65] J. A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

Exp
$$\frac{P \wedge (X[a_1 \cdots a_p.\uparrow^n](e_1,\ldots,e_l) =^?_{\lambda\sigma} \underline{\mathtt{m}}(b_1,\ldots,b_q))}{P \wedge (X[a_1 \cdots a_p.\uparrow^n](e_1,\ldots,e_l) =^?_{\lambda\sigma} \underline{\mathtt{m}}(b_1,\ldots,b_q)) \wedge \bigvee_{r \in R_p \cup R_i} \exists H_1 \ldots \exists H_k : X =^?_{\lambda\sigma} \underline{\mathtt{r}}(H_1,\ldots,H_k) \vee Q}$$
if X is not solved.
where $H_1,\ldots,H_k$ are variables of the appropriate type not occurring in P with the contexts $\Gamma_{H_i} = \Gamma_X$, R_p is the subset of $\{1,\ldots,p\}$ such that $\underline{\mathtt{r}}(H_1,\ldots,H_k)$ has the right type, $R_i =$ if $m \geq n+1$ then $\{m-n+p\}$ else $\emptyset$ and $Q = \exists Y X =^?_{\lambda\sigma} \lambda Y$ if X has a functional type and $Q = \bot$ otherwise.

Exp2
$$\frac{P \wedge (X[a_1 \cdots a_p.\uparrow^n](e_1,\ldots,e_l) =^?_{\lambda\sigma} \lambda b}{P \wedge X[a_1 \cdots a_p.\uparrow^n](e_1,\ldots,e_l) =^?_{\lambda\sigma} \lambda b \wedge \bigvee_{r \in R_p} \exists H_1 \ldots \exists H_k : X =^?_{\lambda\sigma} \underline{\mathtt{r}}(H_1,\ldots,H_k) \vee \exists Y X =^?_{\lambda\sigma} \lambda Y}$$
if X is not solved.
where $H_1,\ldots,H_k$ are variables of the appropriate type not occurring in P with the contexts $\Gamma_{H_i} = \Gamma_X$, R_p is the subset of $\{1,\ldots,p\}$ such that $\underline{\mathtt{r}}(H_1,\ldots,H_k)$ has the right type.

Exp-λ
$$\frac{P}{\exists Y : (A.\Gamma \vdash B), P \wedge X =^?_{\lambda\sigma} \lambda_A Y}$$
if $(X : \Gamma \vdash A \to B) \in \mathcal{TV}ar(P)$, $Y \notin \mathcal{TV}ar(P)$, and X is not a solved variable.

Exp-App
$$\frac{P \wedge X[a_1 \ldots a_p.\uparrow^n] =^?_{\lambda\sigma} \underline{\mathtt{m}}(b_1,\ldots,b_q)}{P \wedge X[a_1 \ldots a_p.\uparrow^n] =^?_{\lambda\sigma} \underline{\mathtt{m}}(b_1,\ldots,b_q) \wedge \bigvee_{r \in R_p \cup R_i} \exists H_1 \ldots \exists H_k, X =^?_{\lambda\sigma} \underline{\mathtt{r}}(H_1,\ldots,H_k)}$$
if X has an atomic type and is not solved.
where $H_1,\ldots,H_k$ are variables of appropriate types, not occurring in P, with the contexts $\Gamma_{H_i} = \Gamma_X$, R_p is the subset of $\{1,\ldots,p\}$ such that $\underline{\mathtt{r}}(H_1,\ldots,H_k)$ has the right type, $R_i =$if $m \geq n+1$ then $\{m-n+p\}$ else $\emptyset$.

Normalise $\dfrac{P \wedge (e_1 =^?_{\lambda\sigma} e_2)}{P \wedge (e_1' =^?_{\lambda\sigma} e_2')}$ if a or b is not in long normal form.
where a' (resp. b') is the long normal form of a (resp. b) if a (resp. b) is not a solved variable and a (resp. b) otherwise.

Replace $\dfrac{P \wedge X =^?_{\lambda\sigma} t}{\{X \mapsto t\}(P) \wedge X =^?_{\lambda\sigma} t}$ if $X \in \mathcal{TV}ar(P), X \notin \mathcal{TV}ar(t)$ and if t is a constant then $t \in \mathcal{TV}ar(P)$.

Table 2. Unification Rules for the $\lambda\sigma$-calculus

Flex-Rig
$$\frac{X(\underline{\mathtt{i}}_1,\ldots,\underline{\mathtt{i}}_\mathtt{m}) =^? \underline{\mathtt{n}}(e_1,\ldots,e_k) :: P}{H_1(\underline{\mathtt{m}},\ldots,\underline{1}) =^? e_1 :: \ldots :: H_k(\underline{\mathtt{m}},\ldots,\underline{1}) =^? e_k :: P\sigma}$$
where $\sigma = \{X/\lambda_{\tau(\underline{\mathtt{i}}_1)} \ldots \lambda_{\tau(\underline{\mathtt{i}}_\mathtt{m})}.\underline{\mathtt{m}+\mathtt{n}}(H_1(\underline{\mathtt{m}},\ldots,\underline{1}),\ldots,H_k(\underline{\mathtt{m}},\ldots,\underline{1}))\}$, $X \notin FV(e_i), \forall i = 1,\ldots,k.$ and $H_1,\ldots,H_k$ are new meta-variables of the appropriate type.

Flex-Flex1
$$\frac{X(\underline{\mathtt{i}}_1,\ldots,\underline{\mathtt{i}}_\mathtt{m}) =^? X(\underline{\mathtt{j}}_1,\ldots,\underline{\mathtt{j}}_\mathtt{m}) :: P}{P\sigma}$$
where $\sigma = \{X/\lambda_{\tau(\underline{\mathtt{i}}_1)} \ldots \lambda_{\tau(\underline{\mathtt{i}}_\mathtt{m})}.H(\underline{\mathtt{k}}_1,\ldots,\underline{\mathtt{k}}_\mathtt{p})\}$,
H is a new meta-variables of the appropriate type and $\{\underline{\mathtt{k}}_1,\ldots,\underline{\mathtt{k}}_\mathtt{p}\} = \{\underline{\mathtt{i}}_\mathtt{s} \mid \underline{\mathtt{i}}_\mathtt{s} = \underline{\mathtt{j}}_\mathtt{s}, \forall s = 1,\ldots m\}$.

Flex-Flex2
$$\frac{X(\underline{\mathtt{i}}_1,\ldots,\underline{\mathtt{i}}_\mathtt{m}) =^? Y(\underline{\mathtt{j}}_1,\ldots,\underline{\mathtt{j}}_\mathtt{n}) :: P}{P\sigma} \text{ if } X \neq Y$$
where $\sigma = \{X/\lambda_{\tau(\underline{\mathtt{i}}_1)} \ldots \lambda_{\tau(\underline{\mathtt{i}}_\mathtt{m})}.H(\underline{\mathtt{k}}_1,\ldots,\underline{\mathtt{k}}_\mathtt{p}), Y/\lambda_{\tau(\underline{\mathtt{j}}_1)} \ldots \lambda_{\tau(\underline{\mathtt{j}}_\mathtt{n})}.H(\underline{\mathtt{k}}_1,\ldots,\underline{\mathtt{k}}_\mathtt{p})\}$ and
$\{\underline{\mathtt{k}}_1,\ldots,\underline{\mathtt{k}}_\mathtt{p}\} = \{\underline{\mathtt{i}}_1,\ldots,\underline{\mathtt{i}}_\mathtt{m}\} \cap \{\underline{\mathtt{j}}_1,\ldots,\underline{\mathtt{j}}_\mathtt{n})\}$.

Table 3. Unification Rules for Patterns