Leopold-Franzens-Universität Innsbruck

Institute of Computer Science
Computational Logic

Higher-Order Unification

Specialisation Seminar
Seminar Report

Daniel Proksch

advised by
assoc. Prof. Dr. René Thiemann

Innsbruck, February 28, 2018

Abstract

We give a general overview over the applications and problems of higher-order unification. Additionally we present and explain an algorithm of unification for higher-order patterns by T. Nipkow. The main goal of the paper is to give a short overview over the topic and explain a possible implementation.

1. Introduction

The notion of unification describes the algorithmic process to solve equations involving symbolic expressions. This is done by applying a substitution to the terms so that the instantiated terms become equal. The general goal is to find a complete and minimal substitution called most general unifier (mgu). Unification is one of the fundamental techniques behind automated reasoning technology. This involves more obvious applications like logic programming or SMT solvers but also some more specific applications like implementations of type inference in dynamic programming languages like Python or Ruby.

The decidability of unification depends on its framework, which is set by the expressions allowed in the unification problem and which expressions are considered equal. First-order syntactical unification, the unification of equal first-order logic terms, is decidable i.e. there is a known algorithm which depending on the terms either states insolubility or gives the most general unifier. That isn't the case for higher-order unification, which is only semi-decidable. This means that the unification process may not halt for unsolvable problems.

2. Motivation for Higher-Order Unification

Even though the extension of terms by quantifiers over expressions isn't as fundamental to most fields of automated reasoning technology as first-order unification, it still has its applications. The λ-calculus is generally just different notation for second- and higher-order logic. So it isn't surprising that higher-order unification plays an important part in extending the general-purpose logic programming language Prolog by adding λ-calculus to achieve λProlog. Another application are generic proof assistants, which use a higher-order version of resolution, e.g. Isabelle.

3. Undecidability of Higher-Order Unification

A decision problem is considered decidable, if there exists an algorithm that always gives a correct true or false value for every possible input. If a decision problem is not decidable, it's considered undecidable. A decision problem is considered semi-decidable, if there exists an algorithm that halts eventually when the value returned is true but may run for ever if the value returned should be false.

As mentioned before, while first-order unification is decidable, that isn't the case for higher-order unification. Without the limitations of first-order logic terms the decision problem becomes semi-decidable and undecidable. The undecidability was proven independently by G. Huet and C.L. Lucchesi in 1972 by reducing Post's correspondence problem. Another, simpler proof was provided by G. Dowek by reducing Hilbert's tenth problem.

4. Unification of Higher-Order Patterns

The algorithm presented is a decidable unification algorithm for higher-order patterns created by T. Nipkow in 1991. It's a simplified version of another algorithm by D. Miller. [2] As notation the algorithm uses terms in λ-calculus, which is a flexible theoretic notation of higher-order logic. The set of higher-order patterns is a subset of all λ-term for which unification is decidable.

4.1 λ-Calculus

Generally the language of λ-terms is defined by following grammar:

$$t \;=\; F \mid x \mid c \mid \lambda x.t \mid (t_1\ t_2)$$

The grammar uses the standard λ-calculus notation, by that t denotes a term, F is a free variable, x represents a bound variable and c is a constant. A free variable is a variable, that isn't bound by a λ-abstraction. Inconveniently the grammar doesn't check whether bound variables are really bound by the lambda abstraction. If this isn't the case, the bound variable becomes a so-called loose bound variable, which should essentially be treated like a constant. Multiple λ-abstractions can be pulled together in a more compact notation.

$$\lambda x.\lambda y.t \;\rightarrow\; \lambda x, y.t$$

There are three kinds of reductions in λ-calculus: The α-conversion, the β-reduction and the η-conversion. If two terms can be β-reduced into the same expression, the terms are β-equivalent. α- and η-equivalence are defined similarly, respectively with α- or η-conversion as reduction.

An α-conversion (also called α-renaming) allows to rename bound variables. Terms that differ only by α-conversion are called α-equivalent. Even though the renaming of bound variables in α-conversion isn't heavily restricted, there are two rules to obey. First, it's not allowed to rename a bound variable so that it gets captured by a different abstraction. Second, α-conversion restricts to only rename variable occurrences, which are bound to the same abstraction. Some possible α-conversionsare presented below.

$$\lambda x.x \;\rightarrow\; \lambda y.y$$
$$\lambda x, x.x \;\rightarrow\; \lambda x, y.y$$

β-reduction represents the application of functions in λ-calculus. β-reducing a λ-term allows to substitute the next element right to a λ-abstraction into the abstraction. Substitutions are in the form $t[x := s]$ with t being the term and everything behind the substitution. To β-reduce a term it's sometimes necessary to α-convert it first. A term that can't be β-reduced further is in β-normal form. Some possible β-reductions are presented below. The equivalence $\equiv$ represents α-conversion is this specific example.

$$(\lambda x.t)\ s\ \rightarrow\ t[x := s]$$
$$(\lambda x.x)\ s\ \rightarrow\ x[x := s]\ =\ s$$
$$(\lambda x.(\lambda x.x)\ x)\ s\ \equiv\ (\lambda x.(\lambda y.y)\ x)\ s\ \rightarrow\ ((\lambda y.y)\ x)[x := s]\ =\ (\lambda y.y)\ s\ \rightarrow\ y[y := s]\ =\ s$$

The remaining reduction type is η-conversion, which is based on the concept of extensional equality, i.e. the concept of stating the equality of objects with the same external properties. In terms of λ-calculus this means that two terms are η-equivalent if and only if the results are equivalent for all possible inputs. A possible η-conversion:

$$\lambda x.(s\ x)\ \rightarrow\ s \quad \text{if and only if} \quad s \text{ does not contain } x$$

4.2 Higher-Order Patterns

As mentioned before, higher-order patterns are λ-terms that behave like first-order terms in unification. This results in the unification for this specific subclass being decidable.

These higher-order patterns are defined as terms t in β-normal form, in which every free variable F is in a subterm $F(\overline{u_n})$ such that the list $\overline{u_n}$ is η-equivalent to a list of distinct bound variables [3]. The term $F(\overline{u_n})$ is a short notation for the iterated application $((\cdots(F\ u_1)\cdots)\ u_n)$, which can also be noted as $F(u_1, \cdots, u_n)$. Generally this means that in higher-order patterns for free variables it's only allowed for bound variables to be applied to them, the application of constants and free variables isn't permitted. Some examples for terms, that are higher-order patterns:

$$F$$
$$\lambda x.F(x)$$
$$\lambda x, y.F(x, y)$$
$$\lambda x, y.(y\ x)$$

Some examples for terms, that aren't higher-order patterns:

$$F\ c$$
$$\lambda x.F(F\ x)$$
$$\lambda x.F(x, x)$$
$$\lambda x, y.F(x, (F\ y))$$

4.3 Unification by Transformation

For his functional unification algorithm T. Nipkow developed a high-level description representing unification of lambda terms as 6 transformation rules. For reasons of complexity and space we will skip the development of those rules into a functional algorithm and focus fully on those transformation rules. The algorithm is only decidable for simply typed lambda calculus.

A unification problem is a problem of determining whether two specified formulas possess a common instance. In the following we present a list of transformation rules of the form $e \longrightarrow \langle E', \theta' \rangle$. These transformation rules will be extended to work on unification problems as follows:

$$\langle e :: E, \theta \rangle \longrightarrow \langle E'@(E\theta' \downarrow_\beta), \theta' \circ \theta \rangle \text{ whenever } e \longrightarrow \langle E', \theta' \rangle$$

When applying those transformations the unification problem E has a solution if and only if $\langle E, \{\} \rangle \rightarrow^* \langle [], \theta \rangle$. The substitution θ describes the most general unifier of E [3].

$$
\begin{aligned}
t =^? t \longrightarrow \quad & \langle [], \{\} \rangle && (1)\\
\lambda x.s =^? \lambda x.t \longrightarrow \quad & \langle [s =^? t], \{\} \rangle && (2)\\
a(\overline{s_n}) =^? a(\overline{t_n}) \longrightarrow \quad & \langle [s_1 =^? t_1, ..., s_n =^? t_n], \{\} \rangle && (3)\\
F(\overline{x_m}) =^? a(\overline{s_n}) \longrightarrow \quad & \langle [H_1(\overline{x_m}) =^? s_1, ..., H_n(\overline{x_m}) =^? s_n], \{F \mapsto \lambda\overline{x_m}.a(\overline{H_n(\overline{x_m})})\} \rangle & \\
& \text{where } F \notin FV(\overline{s_n}) \text{ and } a \text{ is a constant or } a \in \{\overline{x_m}\} && (4)\\
F(\overline{x_n}) =^? F(\overline{y_n}) \longrightarrow \quad & \langle [], \{F \mapsto \lambda\overline{x_n}.H(\overline{z_p})\} \rangle & \\
& \text{where } \{\overline{z_p}\} = \{x_i \mid x_i = y_i\} && (5)\\
F(\overline{x_m}) =^? G(\overline{y_n}) \longrightarrow \quad & \langle [], \{F \mapsto \lambda\overline{x_m}.H(\overline{z_p}), G \mapsto \lambda\overline{y_n}.H(\overline{z_p})\} \rangle & \\
& \text{where } F \neq G \text{ and } \{\overline{z_p}\} = \{\overline{x_m}\} \cap \{\overline{y_m}\} && (6)
\end{aligned}
$$

The equivalence $=^?$ used in the unification problems describes α-, β- and η-equivalence. This enables the usage β-reduction after each transformation to get all terms in the unification problem into β-normal form. This process will always terminate due to our restriction on simply typed lambda calculus. As all α-equivalent terms are assumed to be identified, the usage of α-conversion may also be necessary between transformations. Additionally it's possible to use η-conversion if no transformation rules can be applied otherwise.

The first rule is the removal of trivial equations. It states that two equivalent (in regards to our definition of the used equivalence) terms can be removed without substitution. The application of this rule can be regarded as trivial, depending on the constant application of α- and η-conversion.

$$
\begin{aligned}
t =^? t \quad &\longrightarrow \quad \langle [], \{\} \rangle\\
\langle [\lambda b.F(b) =^? \lambda a.F(a)], \{\} \rangle \quad &\longrightarrow \quad \langle [], \ \{\} \rangle
\end{aligned}
$$

The second transformation rule is the removal of equivalent abstractions. It states that after α-conversion equivalent lambda abstractions can be removed. To achieve this the abstracted bound variables of both terms are named similarly and then regarded to like free variables (but not as free variables), which allows to drop the lambda abstraction.

$$\begin{aligned} \lambda x.s =^? \lambda x.t &\longrightarrow \langle [s =^? t], \{\}\rangle \\ \langle [\lambda a.F(a) =^? \lambda a, b.c(G(b,a))], \{\}\rangle &\longrightarrow \langle [F(a) =^? \lambda b.c(G(b,a))],\ \{\}\rangle \end{aligned}$$

The third rule is the decomposition of rigid-rigid pairs with identical heads. A variables is regarded as rigid if it isn't free in the current term. Decomposition is the process of dropping an application (a function) for its parameters. In an unification problem each parameter is then set equivalent (in regards to our definition of the used equivalence).

$$\begin{aligned} a(\overline{s_n}) =^? a(\overline{t_n}) &\longrightarrow \langle [s_1 =^? t_1, ..., s_n =^? t_n], \{\}\rangle \\ \langle [a(x, \lambda y.c(y)) =^? a(F(z), z)], \{\}\rangle &\longrightarrow \langle [x =^? F(z),\ \lambda y.c(y) =^? z],\ \{\}\rangle \end{aligned}$$

The forth transformation rule is the projection for flexible-rigid pairs. A flexible-rigid pair is an unification problem where one side is an application of n terms on a rigid head, the other one is an application of m bound variables on a free head. This free variable (head) is not allowed to be part of the application on the bound head. This bound variable (head) on the other hand either has to be a constant or part of the application on the free head. The projection then is performed by introducing n new free variables. The free variable (head) then maps by substitution to the n new free variables with the abstracted m bound variables applied onto each of them applied on the rigid head. The remaining equivalences in the unification problem are the m bound variables each applied on the n new free variables equivalent to the corresponding term originally applied to the rigid head.

$$\begin{aligned} F(\overline{x_m}) =^? a(\overline{s_n}) &\longrightarrow \langle [H_1(\overline{x_m}) =^? s_1, ..., H_n(\overline{x_m}) =^? s_n], \\ &\quad \{F \mapsto \lambda\overline{x_m}.a(\overline{H_n(\overline{x_m})})\}\rangle \\ &\quad \text{where } F \notin FV(\overline{s_n}) \text{ and } a \text{ is a constant or } a \in \{\overline{x_m}\} \\ \langle [F(x) =^? c(G(x,y), y)], \{\}\rangle &\longrightarrow \langle [H_1(x) =^? G(x,y),\ H_2(x) =^? y], \\ &\quad \{F \mapsto \lambda x.c(H_1(x), H_2(x))\}\rangle \end{aligned}$$

The fifth rule is the variable elimination for flexible-flexible pairs with identical heads. A flexible-flexible pair with identical heads is a unification problem where both sides are n (potentially) different bound variables on the same free variable (head). This equivalence can be removed completely by mapping the free head to a list of all (in position and variable) equal variables originally applied on the free head, applied on a new free variable and abstracted by one of the two original lists of bound variables.

$$\begin{aligned} F(\overline{x_n}) =^? F(\overline{y_n}) &\longrightarrow \langle [], \{F \mapsto \lambda\overline{x_n}.H(\overline{z_p})\}\rangle \\ &\quad \text{where } \{\overline{z_p}\} = \{x_i \mid x_i = y_i\} \\ \langle [F(x,z) =^? F(x,y)], \{\}\rangle &\longrightarrow \langle [],\ \{F \mapsto \lambda x, z.H(x)\}\rangle \end{aligned}$$

The sixth transformation rule is the variable elimination for flexible-flexible pairs with identical heads with distinct variables. The equation this rule can be applied on is similar to rule five, except the free variables (heads) are distinct. The transformation is similar aswell, except you need an individual substitution per free head which only differs by the abstracted list of bound variables. Those have to match the list originally applied to the free head.

$$\begin{aligned} F(\overline{x_m}) =^? G(\overline{y_n}) \quad &\longrightarrow \quad \langle [], \{F \mapsto \lambda\overline{x_m}.H(\overline{z_p}), G \mapsto \lambda\overline{y_n}.H(\overline{z_p})\}\rangle \\ &\quad \text{where } F \neq G \text{ and } \{\overline{z_p}\} = \{\overline{x_m}\} \cap \{\overline{y_m}\} \\ \langle [G(x) =^? F(x,y)], \{\}\rangle \quad &\longrightarrow \quad \langle [], \ \{F \mapsto \lambda x, y.H(x), \ G \mapsto \lambda x.H(x)\}\rangle \end{aligned}$$

If none of the rules are applicable the unification process fails determining the unification problem as negative. If the algorithm succeeds, the most general unifier is the list of substitutions θ in the unification problem $\langle [], \theta\rangle$ after the process.

As mentioned before, when following these rules a list of pattern unification problems E has a solution if and only if $\langle E, \{\}\rangle \rightarrow^* \langle [], \theta\rangle$. The substitution θ describes the most general unifier of E. The correctness and completeness can be proven by the rules covering all possible cases. Termination on the other hand can be proven by dropping all λ-abstractions and all arguments of free variables turning the resulting set of transformation rules into a variation of first-order unification [3].

The following example shows the application of this algorithm on the unification problem $\langle [\lambda x, y.F(x) =^? \lambda x, z.c(G(z,x))], \{\}\rangle$. β-reduction and α-conversion are applied automatically if needed and the number under the arrow indicates the number of the transformation rule [3].

$$\begin{aligned} &\langle [\lambda x, y.F(x) =^? \lambda x, z.c(G(z,x))], \{\}\rangle && \longrightarrow_2 \\ &\langle [\lambda y.F(x) =^? \lambda z.c(G(z,x))], \{\}\rangle && \longrightarrow_2 \\ &\langle [F(x) =^? c(G(y,x))], \{\}\rangle && \longrightarrow_4 \\ &\langle [H(x) =^? G(y,x))], \{F \mapsto \lambda x.c(H(x))\}\rangle && \longrightarrow_6 \\ &\langle [], \{F \mapsto \lambda x.c(H'(x)), \ H \mapsto \lambda x.H'(x), \ G \mapsto \lambda x, y.H'(x)\}\rangle \end{aligned}$$

5. Outlook

The algorithm presented is not a fully fleshed functional and by that implementable algorithm. Additionally its application is limited to the presented higher-order patterns. While a complete algorithm for higher-order unification can't exist due to its proven undecidability, there are semi decision procedures. One example is Huet's algorithm, which is a semi-decidable algorithm using transformation rules to perform a systematic search of potential unifiers. Just like T. Nipkow's algorithm presented in this report, Huet's algorithm works well in practice [1].

Due to that it's safe to say, that the undecidability of higher-order unification doesn't fully prevent the practical usage of it. While due to its semi-decidability full implementations may not terminate, certain algorithms still allow the implementation for specific cases or with certain constraints. Currently the usage of higher-order unification seems to be limited to proof assistants or other logic related applications, but it seems possible that other, more general application fields might be in need of it in the future.

References

[1] Gilles Dowek. Handbook of automated reasoning. chapter Higher-order Unification and Matching, pages 1009–1062. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[2] Tobias Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349. IEEE Press, 1991.

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