# Type Independent Unification of Higher-Order Patterns

Jean-Pierre Jouannaud

Université Paris-Saclay, Laboratoire de Méthodes Formelles, 91190 Gif-sur-Yvette, France.

## 1 Introduction

Unification is used to show confluence of a set of overlapping rewrite rules. We are interested here in higher-order rules whose left-hand sides are patterns, as introduced by Miller in the case of simple types [9], later considered for other typing disciplines [2], and finally adapted to untyped lambda calculi in [3]. Following up, we study here unification of patterns *independently of a given typing discipline*. To this end, we introduce the notion of *typed structure* by axiomatizing sets of typable terms without any syntactic notion of type, as subsets of untyped terms that satisfy some well crafted closure properties common to most type systems. We then show that typed structures enjoy most general unifiers for patterns, computable by the usual algorithm for unifying patterns. This axiomatization is somehow reminiscent of Girard's axiomatisation of typed structures on which strong normalization proofs of typed lambda calculi are based. We finally briefly discuss the associative-commutative case.

Typed rewriting structures, their unifiability properties, and their application to checking higher-order confluence are developed in [5], where the main result described here is proved.

## 2 Computations on higher-order terms

Given an *untyped* lambda calculus generated by a vocabulary made of three pairwise disjoint sets, a signature $\mathcal{F}$ of *function symbols*, a set $\mathcal{X}$ of *variables*, and a set $\mathcal{Z}$ of *meta-variables*, we are interested in $\lambda\mathcal{F}$, an untyped calculus whose reduction relation extends the $\beta$-rule of the underlying lambda calculus by a set of user-defined rewrite rules built over that vocabulary.

### 2.1 Terms

$\lambda\mathcal{F}$ is a mix of the pure lambda calculus and Klop's combinatory reduction systems [7]. Terms are those of the lambda calculus enriched with $\mathcal{F}$-*headed* terms of the form $f(\overline{u})$ with $f \in \mathcal{F}$, $\overline{u}$ denoting a list of terms separated by commas, and *meta-terms* of the form $Z[\overline{v}]$ with $Z \in \mathcal{Z}$. Only variables can be abstracted over. Elements of the vocabulary have arities, denoted by vertical bars as in $|f|$. Variables have arity zero, meta-variables have an arbitrary arity. The set of (open) terms, $\mathcal{T}_{\lambda\mathcal{F}}$, is defined by the following grammar rules:

$$u, v := x \mid (u\ v) \mid \lambda x.v \mid f(\overline{u}) \mid Z[\overline{v}]$$
$$\text{where } x \in \mathcal{X},\ f \in \mathcal{F},\ |\overline{u}| = |f|,\ Z \in \mathcal{Z} \text{ and } |\overline{v}| = |Z|$$

We write $a$ for $a(\,)$, $X$ for $X[\,]$ and $f(x\ y)$ for $f((x\ y))$. We use the small letters $f, g, h, \ldots$ for function symbols, $x, y, z, \ldots$ for variables, and reserve capital letters $X, Y, Z, \ldots$ for meta-variables. When convenient, a small letter like $x$ may denote any variable in $\mathcal{X} \cup \mathcal{Z}$. By function symbols we sometimes mean those in $\mathcal{F}$, as well as application and abstraction.

We use the notation $|\_|$ for various quantities besides symbols arities (length of lists, size of expressions, the cardinality of sets), and $[m..n]$ for the list of natural numbers from $m$ to $n$.

Positions in higher-order terms, as in first-order terms, are words over the natural numbers, using $\Lambda$ for the empty word, $\cdot$ for concatenation, $\leq_{\mathcal{P}}$ for the prefix order (*above*), $\geq_{\mathcal{P}}$ for its inverse (*below*), $<_{\mathcal{P}}$ and $>_{\mathcal{P}}$ for their strict parts, $p\#q$ for incomparable positions (*parallel*), and $p \geq_{\mathcal{P}} Q$ ($p \leq_{\mathcal{P}} Q$, resp.), $Q$ a set of parallel positions, for $\exists q \in Q : \ p \geq_{\mathcal{P}} q$ ($p \leq_{\mathcal{P}} q$, resp.).

Given a term $M$, we use: $M(p)$ for its symbol at position $p$; $M|_p$ for the subterm of $M$ at position $p$, a notion which is sometimes convenient and will be given a precise meaning later; $\mathcal{P}os(M)$, $\mathcal{FP}os(M)$, $\mathcal{VP}os(M)$, $\mathcal{MP}os(M)$ for the following respective sets of positions of $M$: all positions, the positions of function symbols, of free variables, and of meta-variables; $\mathcal{V}ar(M)$ for its sets of free variables; $\mathcal{MV}ar(M)$, $\mathcal{MV}ar^{l}(L)$ and $\mathcal{MV}ar^{nl}(M)$ for its sets of arbitrary, linear and non-linear meta-variables; A term $M$ is *ground* if $\mathcal{V}ar(M) = \emptyset$, *closed* if $\mathcal{MV}ar(M) = \emptyset$, and *linear* if $\mathcal{MV}ar^{nl}(M) = \emptyset$. We use $\mathcal{T}$ for the set of closed terms.

## 2.2 Substitutions

A *substitution* is a map from variables and meta-variables to terms which extends to a *capture-avoiding* homomorphism on terms [7]. The result $t\sigma$ of substituting the term $t$ by the substitution $\sigma$ is called an *instance* of $t$. All substitutions considered here will have a finite domain, hence can be denoted in extension as in $\{x_1 \mapsto M_1, \ldots x_n \mapsto M_n\}$ or $\{\overline{x} \mapsto \overline{M}\}$, where $\overline{x}$ is a list of variables or meta-variables. The substitution $\sigma$ is *ground* (resp., *closed*) when so are all $M_i$'s. The *domain* of $\sigma$ is the set $\mathcal{D}om(\sigma) = \{x_i : \sigma(x_i) \neq x_i\}_i$ while $\mathcal{R}an(\sigma) = \bigcup_{x \in \mathcal{D}om(\sigma)} \mathcal{V}ar(\sigma(x)) \cup \mathcal{MV}ar(\sigma(x))$ is its *image*. A substitution $\sigma$ can be *restricted to* or *deprived from* (meta-)variables in some set $V$, written $\sigma_{|V}$ and $\sigma_{\setminus V}$ respectively. We denote by $\mathcal{P}os(\sigma)$ the sequence $\{\mathcal{P}os(\sigma(x_i))\}_i$ of sets of positions of $\sigma$.

## 2.3 Splitting and sticking

Given a term $u$ and a list $P = \{p_i\}_{i=1}^{i=n}$ of parallel positions in $u$, we define the term obtained by *splitting* $u$ along $P$ as $\underline{u}_P = u[Z_1(\overline{x_1})]_{p_1} \ldots [Z_n(\overline{x_n})]_{p_n}$ ($u$ is cut below $P$) and its associated substitution by $\overline{u}^P = \{Z_i \mapsto \lambda\overline{x_i}.u|_{p_i}\}_{i=1}^{i=n}$ ($u$ is cut above $P$), where, for all $i \in [1, n]$, $\overline{x_i}$ is the list of all variables of $u|_{p_i}$ bound in $u$ above $p_i$ and $Z_i$ is a fresh meta-variable of arity $|\overline{x_i}|$. The definition of substitution for meta-variables ensures that $\overline{u}^P\underline{u}_P = u$. Instantiating $\underline{u}_p$ by $\overline{u}^p$ amounts therefore to stick $u|_p$ in the *hole* of the *context* $u[\_]_p$, an operation that may capture free variables of $u|_p$: splitting gives a meaning for the operation of *sticking* a term inside another in terms of the familiar substitution operation. (Sticking is usually called *replacement* when no variable is captured.) We denote by $u[\_]_P$ a context with holes at a set $P = \{p_i\}$ of parallel positions in $u$, and by $u[\overline{v}]_P$ the term obtained by sticking each $v_i$ at position $p_i$ in $u$. The brackets used in contexts may sometimes collide with those used for meta-variables, requiring desambiguation by the user.

## 2.4 Reductions

Two different kinds of reductions coexist in $\lambda\mathcal{F}$, functional and higher-order reductions, both operating on closed terms. However, rewriting open terms will sometimes be needed, in which case rewriting is intended to rewrite all their closed instances at once.

## 2.5 Functional reductions

*Functional reduction* is the relation on terms generated by the rule $\beta_\alpha : (\lambda x.v\ w) \longrightarrow v\{x \mapsto w\}$. The usually omitted $\alpha$-index stresses that renaming bound variables, called $\alpha$-*conversion*, is built-in, that is, rewriting with $\beta_\alpha$ is *modulo* $\alpha$-conversion (only those variables bound below the rewriting position need be renamed).

## 2.6 Higher-order reductions

*Higher-order reductions* result from rules whose left-hand sides are higher-order patterns in Miller's or Nipkow's sense [8], although they need not be typed here.

**Definition 1** (Untyped pattern)**.** *A* pre-redex *of arity $n$ in a term $L$ is an unapplied meta-term $Z[\overline{x}]$ whose arguments $\overline{x}$ are $n$ pairwise distinct variables. A* pre-pattern *is a ground term all of whose meta-variables occur in pre-redexes. An* untyped pattern*, or simply* pattern*, is a pre-pattern which is neither a pre-redex nor an abstraction.*

Note that erasing types from a Nipkow's pattern yields a pattern in our sense, since his pre-redexes being of base type, they cannot be applied. Observe that pre-redexes in pre-patterns can only occur at parallel positions.

We can now define higher-order rules and rewriting:

**Definition 2** (Rule)**.** *A (higher-order)* rule *is a triple $i : L \to R$, whose (possibly omitted)* index *$i$ is a name, left-hand side $L$ is a pattern, and $\mathcal{MV}ar(R) \subseteq \mathcal{MV}ar(L)$.*

The use of capital letters for higher-order rules aims at pointing out that $L, R$ are higher-order terms, that is, are built using the abstraction and application operators and meta-variables of arity at least one. In contrast, first-order terms have no abstractions, no applications, and no meta-variables of non-zero arity. We will use small letters for them, as in $l \to r$.

The $\beta$-reduction rule is a particular case of higher order rule written $(\lambda x.X[x]\ u) \to X[u]$.

**Definition 3** (Higher-order rewriting)**.** *Given an open term $u$, a position $p \in \mathcal{P}os(u)$, and a rule $i : L \to R$, $u$ rewrites with $i$ at $p$, written $u \xrightarrow[i]{p} v$, iff $u|_p = L\gamma$ for some substitution $\gamma$, and $v = u[X[\overline{x}]]_p\{X \mapsto \lambda\overline{x}.R\gamma\} = u[R\gamma]_p$, where $\overline{x}$ is the list of variables of $u|_p$ which are bound above the position $p$ in $u$. We write $u \xrightarrow[\mathcal{R}]{p} v$ for $\exists i \in \mathcal{R} : u \xrightarrow[i]{p} v$.*

*A $\lambda\mathcal{F}$-*rewrite system *is a pair $(\mathcal{F}, \mathcal{R})$ made of a user's signature $\mathcal{F}$ and a set $\mathcal{R}$ of higher-order rewrite rules on $\mathcal{F}$ containing beta, defining the rewrite relation of $\lambda\mathcal{F}$ as $\xrightarrow[\mathcal{R}]{}$.*

$\lambda\mathcal{F}$-rewrite systems are being used in a variety of proof assistants, notably in Agda, Isabelle, Dedukti, and Coq. As a higher-order rewriting format, $\lambda\mathcal{F}$ is a Combinatory Reduction System [10]. This is not surprising since all other known higher-order rewriting formats can be encoded as Combinatory Reduction Systems [11].

# 3 Typed rewriting structures

The role of typing is to characterize subsets of the set of higher-order closed terms that satisfy good properties for computing. Calling generically $\mathcal{TT}$ such subsets of closed terms, its elements are called *typed closed terms*. Computations are meant to operate on typed closed terms, but rewriting is based on open terms, that is terms with meta-variables.

We denote therefore by $\mathcal{TT}_{\lambda\mathcal{F}}$ the set of *typed open terms*, assuming $\mathcal{TT} \subseteq \mathcal{TT}_{\lambda\mathcal{F}}$. In order to dispense with explicit types, we say that a typed open substitution $\sigma$ is *well-typed* for a typed open term $u$ if $u\sigma$ is a typed open term, and write $\sigma \in \mathcal{TT}_{\lambda\mathcal{F}}(u)$. More generally, $\sigma$ is *well-typed* for $\theta$ if $\sigma$ is well-typed for all $u$ such that $x \mapsto u \in \theta$. Splitting allows then to define whether the replacement of a subterm at a particular position by a typed term yields a typed term: we define $u \in \mathcal{TT}(w[\_]_p)$ iff $w[u]_p \in \mathcal{TT}_{\lambda\mathcal{F}}$, that is, iff $\{Z \mapsto \lambda\overline{z}.u\} \in \mathcal{TT}_{\lambda\mathcal{F}}(w[Z[\overline{z}]]_p)$. We omit mention of "open/closed" when it matters not or can be inferred from the context.

The *axioms* that a *typed structure* $\mathcal{TT}_{\lambda\mathcal{F}}$ such that $\mathcal{X} \cup \mathcal{Z} \subseteq \mathcal{TT}_{\lambda\mathcal{F}} \subseteq \mathcal{T}_{\lambda\mathcal{F}}$ should satisfy (and that $\mathcal{T}_{\lambda\mathcal{F}}$ itself satisfies) are the following:

H0 $\mathcal{TT}_{\lambda\mathcal{F}}$ is closed under $\alpha$-conversion, and renaming of a free occurrence of a given variable (without capture) or meta-variable;

H1 abstraction: $u \in \mathcal{TT}_{\lambda\mathcal{F}}$ iff $\lambda z.u \in \mathcal{TT}_{\lambda\mathcal{F}}$;

H2 composition: $\sigma \in \mathcal{TT}_{\lambda\mathcal{F}}(u)$, $\tau \in \mathcal{TT}_{\lambda\mathcal{F}}(\sigma)$ and $\mathcal{D}om(\tau) \cap (\mathcal{V}ar(u) \cup \mathcal{MV}ar(u)) = \varnothing$ implies $\tau \in \mathcal{TT}_{\lambda\mathcal{F}}(u\sigma)$;

H3 splitting: $u\sigma \in \mathcal{TT}_{\lambda\mathcal{F}}$ implies $u \in \mathcal{TT}_{\lambda\mathcal{F}}$ and $\sigma \in \mathcal{TT}_{\lambda\mathcal{F}}(u)$;

H4 patterns: if $L$ is a pattern, then $L \in \mathcal{TT}_{\lambda\mathcal{F}}$.

Interpreting membership to $\mathcal{TT}_{\lambda\mathcal{F}}$ by Curry-style typability for some typing context, sets of typed terms satisfy these assumptions for all usual type systems that have the unique type property.

Typed structures enjoy a few more important closure properties, notably subterm, monotonicity, stability, as well as unifiability:

**Lemma 1** (Unifiability)**.** *Let $u, v$ be two terms unifiable by a substitution $\sigma$ such that $u\sigma = v\sigma$ is well-typed, that is, $u\sigma \in \mathcal{TT}_{\lambda\mathcal{F}}$. Then, $\forall w[\_]$ such that $(\mathcal{V}ar(w[\_]) \cup \mathcal{MV}ar(w[\_])) \cap \mathcal{D}om(\sigma) = \varnothing$, $u \in \mathcal{TT}_{\lambda\mathcal{F}}(w[\_])$ iff $v \in \mathcal{TT}_{\lambda\mathcal{F}}(w[\_])$.*

## 4 Unification in typed rewriting structures

We now investigate a major property of typed rewriting structures, the existence of most general unifiers for solvable *critical pair equations*, that is, equations $U = V$ such that one of $U, V$ is a left-hand side of rule, and the other a subterm of a left-hand side of rule. In other words, if the equation $U = V$ is unifiable in the untyped world, then it is unifiable in a typed structure. Further, the most general unifier of the untyped structure happens to belong to any typed structure, hence must be most general in any typed structure.

**Definition 4.** *A* unification (equational) problem *is a conjunction of elementary equations. An* elementary equation *is either the constant $\perp$ or is of the form $u = v$ in which $u$ is a pre-pattern and $v$ is a pre-pattern.*

A set of transformation rules for higher-order unification of untyped patterns is described in [4] for linear patterns and meta-variables having a bounded arity, and its extension to non-linear ones is also sketched, following the standard path by adding a *Merge* rule. These unification rules are recalled in Figure 1. They are essentially those for simply typed patterns [9], see also [6]. As usual, the rules transform elementary equations into a conjunction thereof until some *solved form* is eventually obtained. They use the following definition:

**Definition 5.** *A free variable $x \in \mathcal{X}$ is* protected *in a pre-pattern $u$ if all its occurrences in $u$ belong to a pre-redex of $u$. We denote by $\mathcal{UV}ar(u)$ the set of unprotected variables of $u$.*

| | | | | |
|---|---|---|---|---|
| *Dec-Fun* | $f(\overline{u}) = f(\overline{v})$ | $\longrightarrow$ | $\bigwedge_{i=1}^{i=\lvert f\rvert} u_i = v_i$ | if $f \in \mathcal{F} \cup \mathcal{Z} \cup \{@\}$ |
| *Dec-Abs* | $\lambda x.u = \lambda y.v$ | $\longrightarrow$ | $u\{x \mapsto z\} = v\{y \mapsto z\}$ | with $z$ fresh |
| *Merge* | $X[\overline{x}] = u \wedge X[\overline{y}] = v$ | $\longrightarrow$ | $X[\overline{z}] = u\{\overline{x} \mapsto \overline{z}\} \wedge u\{\overline{x} \mapsto \overline{z}\} = v\{\overline{y} \mapsto \overline{z}\}$ | if $\lvert u\rvert \leq \lvert v\rvert$ |
| *Swap* | $u = Y[\overline{y}]$ | $\longrightarrow$ | $Y[\overline{y}] = u$ | if $u$ is not a pre-redex |
| *Flip* | $X[\overline{x}] = Y[\overline{y}]$ | $\longrightarrow$ | $Y[\overline{y}] = X[\overline{x}]$ | if $\lvert X\rvert - \lvert\overline{x}\rvert > \lvert Y\rvert - \lvert\overline{y}\rvert$ |
| *Drop* | $X[\overline{x}] = u[Y[\overline{y}]]_q$ | $\longrightarrow$ | $X[\overline{x}] = u[Z[\overline{z}]]_q \wedge Y[\overline{y}] = Z[\overline{z}]$ | |

where $\overline{z} = \overline{y} \cap (\overline{x} \cup \mathcal{BV}ar(u))$ and $Z$ fresh s.t. $\lvert Z\rvert = \lvert Y\rvert - \lvert\overline{y}\rvert + \lvert\overline{z}\rvert$,
if $\overline{y} \not\subseteq \overline{x} \cup \mathcal{BV}ar(u)$, $\lvert X\rvert = \lvert\overline{x}\rvert$ if $u(\Lambda) \in \mathcal{F} \cup \{@, \lambda\}$, $\mathcal{UV}ar(u) \subseteq \overline{x}$
and $\lvert Y\rvert - \lvert\overline{y}\rvert \geq \lvert X\rvert - \lvert\overline{x}\rvert$ if $q = \Lambda$,
where $\mathcal{UV}ar(u)$ denotes the set of variables of $u$ whose one occurrence does not occur in an argument of a meta-variable.

Figure 1: Non-failure unification rules for equational problems

For an example, $x$ is protected in $f(g(X[x]), X)$, but not in $f(g(X[x]), x)$ because of its second occurrence. Protected variables can be eliminated from a term by appropriately instantiating its meta-variables as done in the *Drop* rule.

The first rule of Figure 1 is the same as that for first-order unification. *Dec-Abs* is the particular case for abstractions. *Merge* eliminates all occurences of a non-linear meta-variable but one. *Swap*, *Flip* and *Drop* put the equations in a format appropriate for extracting the most general unifier.

The rules of Figure 1 suffice when a unification problem is known to be solvable, otherwise failure rules are also needed to detect non-unifiability. These rules are recalled in Figure 2. An important known observation exploited in rule *Fail-Protect* is that elementary unification problems for which a free variable occurs unprotected on one side, and does not occur at all on the other side, have no solution.

**Theorem 1.** *Assume $\sigma$ is a well-typed unifier for some critical pair equation $U = V$. Then, $mgu(U{=}V) \in \mathcal{TT}_{\lambda\mathcal{F}}(U, V)$. It is obtained by applying the rules of Figure 1 until no more possible. Non-unifiability of an equational problem $P_0$ is obtained when the whole set of rules fails, that is, returns the constant $\perp$.*

Therefore, unifiability of typed patterns, and the expression of a most general unifier, does not depend upon a particular Curry-style type system for the lambda calculus, provided that the type system satisfies our axioms. This new result was already observed in particular cases.

Once soundeness of the rules is proved, the proof given in [5] is based on the preservation by the unification rules of Figure 1 of an appropriate invariant expressing that some substitution is a solution of the starting unification problem. In case no solution is known, the proof relies on termination of the whole set of unification rules, which must therefore end up with an application of *global-Failure* in case no solution exists to the starting unification problem.

| | | | | |
|---|---|---|---|---|
| *Conflict* | $f(\overline{u}) = g(\overline{v})$ | $\longrightarrow$ | $\perp$ | if $f, g \in \mathcal{F} \cup \mathcal{X} \cup \{@, \lambda\}$ and $f \neq g$ |
| *Fail-Protect* | $X[\overline{x}] = u$ | $\longrightarrow$ | $\perp$ | if $\exists z {\in} \mathcal{UV}ar(u) \setminus \overline{x}$ |
| *Global-Failure* | $P \wedge \perp$ | $\longrightarrow$ | $\perp$ | |

Figure 2: Failure unification rules

Let us illustrate some rules, using meta-variables according to our convention.

$$f(\lambda z.X[z]) = f(\lambda z.z) \underset{Dec-Fun}{\longrightarrow} \lambda z.X[z] = \lambda z.z \underset{Dec-Abs}{\longrightarrow} X[z] = z$$
$$f(\lambda z.X) = f(\lambda z.z) \underset{Dec-Fun}{\longrightarrow} \lambda z.X = \lambda z.z \underset{Dec-Abs}{\longrightarrow} X = z \underset{\textit{Fail-Protect}}{\longrightarrow} \bot$$
$$f(\lambda y.f(U)) = f(X) \underset{Dec-Fun}{\longrightarrow} \lambda y.f(U) = X \underset{\textit{Swap}}{\longrightarrow} X = \lambda y.f(U) \underset{Meta-Abs}{\longrightarrow} X[y] = f(U)$$
$$f(Y) = f(\lambda y.f(U)) \underset{Dec-Fun}{\longrightarrow} Y = \lambda y.f(U) \underset{Meta-Abs}{\longrightarrow} Y[y] = f(U) \underset{\textit{Fail-Arity}}{\longrightarrow} \bot$$
$$T = \lambda y.Y[y] \underset{Meta-Abs}{\longrightarrow} T[y] = Y[y] \underset{\textit{Flip}}{\longrightarrow} Y[y] = T[y]$$
$$Y[z] = \lambda x.T[y,z] \underset{Meta-Abs}{\longrightarrow} Y[z,x] = T[y,z] \underset{\textit{Drop}}{\longrightarrow} Y[z,x] = Z[z] \wedge T[y,z] = Z[z]$$

(where $Z$ is a fresh variable of arity 1)

Since Associativity and Commutativity define a syntactic theory whose unification algorithm can be expressed by rewrite rules [1], we conjecture that AC-unification of higher-order patterns does not depend either upon a particular type system satisfying our axioms. Whether these result also extend to type systems having a principal type instead of a unique type is also open.

## References

[1] Alexandre Boudet and Evelyne Contejean. ”syntactic” ac-unification. In Jean-Pierre Jouannaud, editor, *Constraints in Computational Logics, First International Conference, CCL'94, Munich, Germany, September 7-9, 1994*, volume 845 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 1994.

[2] Gilles Dowek. Higher-order unification and matching. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1009–1062. Elsevier and MIT Press, 2001.

[3] Gilles Dowek, Gaspard Férey, Jean-Pierre Jouannaud, and Jiaxiang Liu. Confluence of left-linear higher-order rewrite theories by checking their nested critical pairs. *Mathematical Structures in Computer Science*, Special issue on Confluence:1–36, 2022.

[4] Gaspard Férey and Jean-Pierre Jouannaud. Confluence in UnTyped Higher-Order Theories by Means of Critical Pairs. draft hal-03126102, INRIA, march 2021.

[5] Jean-Pierre Jouannaud. Confluence in terminating rewriting computations. In Bertrand Meyer, editor, *The French School of Programming*. Springer Verlag, 2024.

[6] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT-Press, 1991.

[7] Jan Willem Klop. *Combinatory Reduction Systems*. Number 127 in Mathematical Centre Tracts. CWI, Amsterdam, The Netherlands, 1980. PhD Thesis.

[8] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[9] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[10] Terese. Term rewriting systems. In *Cambridge Tracts in Theoretical Computer Science, M. Bezem, J. W. Klop, and R. de Vrijer editors*. Cambridge University Press, 2003.

[11] Vincent van Oostrom and Femke van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In Jan Heering, Karl Meinke, Bernhard Möller, and Tobias Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting, First International Workshop, HOA '93, Amsterdam, The Netherlands, September 23-24, 1993, Selected Papers*, volume 816 of *Lecture Notes in Computer Science*, pages 276–304. Springer, 1993.