Generic Second-Order Matching, Higher-Order Preunification and Pattern Unification - Implementations in Haskell

Nikolai Kudasov, Artem Starikov, Fedor Ivanov, Damir Afliatonov

▶ **To cite this version:**

Nikolai Kudasov, Artem Starikov, Fedor Ivanov, Damir Afliatonov. Generic Second-Order Matching, Higher-Order Preunification and Pattern Unification - Implementations in Haskell. UNIF 2025 - 39th International Workshop on Unification, Jul 2025, Birmingham, United Kingdom. ⟨hal-05148806⟩

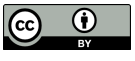
HAL Id: hal-05148806

https://hal.science/hal-05148806v1

Submitted on 7 Jul 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Generic Second-Order Matching, Higher-Order Preunification and Pattern Unification Implementations in Haskell

Nikolai Kudasov[1], Artem Starikov[1], Fedor Ivanov[1], and Damir Afliatonov[1]

Innopolis University, Innopolis, Russia
n.kudasov@innopolis.ru
a.starikov@innopolis.university
f.ivanov@innopolis.university
d.afliatonov@innopolis.university

Higher-order unification (HoU) is a process of solving symbolic equations with functional variables and has applications in automated theorem proving, type systems implementations, and higher-order logic programming. In dependent type systems, HoU is crucial [19] to enable implicit arguments [24].

In the literature, HoU algorithms are almost always exclusive to variations of λ-calculus. This is usually justified by appealing to higher-order abstract syntax (HOAS) [23]. However, it is unclear if encoding terms of a given language to HOAS and then decoding HoU solutions is always practical. Importantly, it seems that *generic* programming is not feasible with such representation [11].

Dependent languages such as Agda [21], Idris [4], Rocq [3], and Lean [5] all implement at least a version of higher-order pattern unification [20, 1] specialized to their respective languages and term representations (not HOAS). Implementing correct and efficient HoU algorithms takes a lot of effort [10, §5.7]. Unfortunately, existing implementations are not reusable, raising the barrier for developing new dependently typed languages or pushing severe limitations on their users[1]. In particular, without implicit arguments, many dependently typed programs quickly become overly verbose.

Generic algorithms require generic syntax representation. Data types à la carte [26] offer one such representation as a solution to the Expression Problem [28], where a recursive data type definition is split into a non-recursive type that describes possible nodes and a general recursive structure. This allows for generic grafting[2] and efficient generic implementations of algorithms such as first-order unification [25]. The `hypertypes`[3] project extends the approach to support for mutually recursive structures, and, building on generic first-order unification, offers a form of generic Hindley-Milner type inference[4]. Free scoped monads [14] is another variation of data types à la carte, supporting syntax with binders and providing generic capture-avoiding substitution. The latter also has an experimental untyped generalized Huet's higher-order preunification [9].

One approach to make HoU algorithms truly language-agnostic is to rely on second-order abstract syntax (SOAS) [8]. In fact, HoU problems can be seen as E-unification problems for SOAS [12]. Unfortunately, since most HoU algorithms rely explicitly on λ-calculus and its syntactic properties, properly generalizing these algorithms to SOAS poses a challenge. For example, most HoU algorithms distinguish "flex" and "rigid" terms, based on whether the "head"

[1] For instance, experimental type theory implementations such as `cubicaltt`, **cooltt**, **redtt**, Rzk either avoid HoU entirely or choose a simple algorithm to avoid complicating the implementation.

[2] *Grafting* is substitution that is ignorant of name captures.

[3] https://github.com/lamdu/hypertypes

[4] See `Hyper.Infer` module.

of the term is a metavariable. Thus, generalizing these algorithms requires generalization of the corresponding notions. While partial generalization attempts exist [13], a proper generalization is yet to be developed.

In this work, we rely on "Free Foil" [16], a more efficient[5] variation of free scoped monads, to implement generic and reasonably efficient language-agnostic HoU algorithms in Haskell. "Free Foil" also offers Template Haskell support, which significantly speeds up prototyping of small languages. The following sections provide some details for the current work-in-progress. The Haskell implementation is available on GitHub at github.com/fedor-ivn/free-foil-hou.

1 Second-Order Abstract Syntax via Free Foil

The generic variations of HoU algorithms discussed in this work are designed around second-order abstract syntax (SOAS) [8], where terms are either (regular) *variables* (x, y, z), *parametrized metavariables* ($\mathrm{M}[t_1, t_2, \ldots, t_n]$), or *operators* (i.e. *functional symbols* or *constants*) that may have regular subterms as well as subterms with local scopes (binding regular variables).

In this work, SOAS is implemented via Free Foil, which provides everything except the parameterized metavariables. Specifically, Free Foil provides the `AST` datatype, indexed by the type of binders (to support different syntax for binding variables), language signature, and a phantom type parameter `n` representing the scope that the term of type `AST binder sig n` belongs to. The scope parameter is crucial in the Foil technique [18] that makes the efficient underlying representation typesafe.

```
-- Type of terms in an abstract syntax generated from a signature sig,
-- with binders determined by binder, and having free variables in scope n.
data AST binder sig n where
  Var :: Foil.Name n -> AST binder sig n
  Node :: sig (ScopedAST binder sig n) (AST binder sig n) -> AST binder sig n

-- | Scoped term (bound variables + term that is may use those variables).
data ScopedAST binder sig n where
  ScopedAST :: binder n l -> AST binder sig l -> ScopedAST binder sig n
```

For example, a signature for untyped lambda calculus may be defined as follows:

```
-- Signature for the terms of untyped lambda calculus
data TermSig scoped term
    = Lam scoped      -- λ(x.t)
    | App term term   -- t1 t2

-- NameBinder is the default kind of pattern (introducing exactly one variable)
type LambdaTerm = AST NameBinder TermSig
```

For SOAS, metavariables are added using data types à la carte approach [26, 15], immediately parameterizing over the type of metavariable identifiers:

```
-- Signature for parametrized metavariables
data MetaAppSig metavar scope term = MetaAppSig metavar [term]
```

[5] The underlying representation follows the "rapier" [22] which is used by GHC and Agda.

```
-- Second-Order Abstract Syntax over a given signature
type SOAS metavar sig n = AST NameBinder (Sum sig (MetaAppSig metavar)) n
```

For type-directed unification, SOAS terms need to be annotated with type information. The `TypedSOAS` type specifies such annotations for all subterms (except regular variables):

```
data AnnSig ann sig scopedTerm term = AnnSig (sig scopedTerm term) ann
data AnnBinder ann binder (n :: Foil.S) (l :: Foil.S)
  = AnnBinder (binder n l) ann

type TypedSOAS typ metavar binder sig n =
  AST (AnnBinder typ binder) (AnnSig typ (Sum sig (MetaAppSig metavar))) n
```

These and other definitions in the implementation rely on generalized algebraic data types, which is a commonly used Haskell extension, also available in many other typed functional programming languages. As such, the technique can be relatively easily ported to other languages.

2 Matching for SOAS

Matching is a restricted case of unification where only metavariables in the left-hand side of the equation are considered for substitution [6, Section 6.6]. Matching can be viewed as an instance of unification where the right hand side does not have metavariables (or those are not subject to substitutions). Formally, given terms t_1 (containing metavariables to be instantiated) and t_2 (ground), matching finds a substitution θ such that $\theta t_1 \equiv_\alpha t_2$.

The transition rules here and in the next section are presented following the notation introduced in [12, Section 4.2]: each transition rule is written in the form

$$c \xrightarrow{\theta} \{c_1, c_2, \ldots, c_n\}$$

where

- c is a matching constraint (picked non-deterministically from the set of all constraints) of the form $\Theta \mid \Gamma \vdash t \stackrel{?}{=} u : \tau$
  - Θ is the metavariable context (list of metavariables with their types)
  - Γ is the regular variable context (list of variables with their types)
  - t is a term, possibly containing parametrised metavariables;
  - u is the rigid term, without any metavariables;
  - τ is the type of both t and u in the given context $\Theta \mid \Gamma$
- $\{c_1, c_2, \ldots, c_n\}$ is a (possibly empty) set of constraints that replaces constraint c during the transition step
- θ is the metavariable substitution (a partial solution) corresponding to the transition rule; the full solution is the composition of such substitutions for the chain of transition rules that starts with the initial set of constraints and ends with an empty set of constraints.

Here are the transition rules of the proposed matching algorithm:

$$(\Theta \mid \Gamma \vdash x \stackrel{?}{=} x : \tau) \xrightarrow{\mathsf{id}} \varnothing, \text{where } (x : \tau) \in \Gamma \qquad \textsf{(delete)}$$

$$(\Theta \mid \Gamma \vdash \mathsf{F}(\overline{\overline{x}.s}) \stackrel{?}{=} \mathsf{F}(\overline{\overline{x}.t}) : \tau) \xrightarrow{\mathsf{id}} \{\Theta \mid \Gamma, \overline{x_i} : \overline{\sigma_i} \vdash s_i \stackrel{?}{=} t_i : \tau_i\}^{i \in \{1,\dots,n\}} \qquad \textsf{(decompose)}$$

$$(\Theta \mid \Gamma \vdash \mathrm{M}[\overline{s}] \stackrel{?}{=} u : \tau) \xrightarrow{[\mathrm{M}[\overline{x}] \mapsto z_i]} \{\Theta \mid \Gamma \vdash s_i \stackrel{?}{=} u : \tau\} \qquad \textsf{(project}_i\textsf{)}$$

$$(\Theta \mid \Gamma \vdash \mathrm{M}[\overline{s}] \stackrel{?}{=} \mathsf{F}(\overline{\overline{x}.t}) : \tau) \xrightarrow{[\mathrm{M}[\overline{x}] \mapsto \mathsf{F}(\overline{\overline{x}.\mathrm{T}[\overline{x},\overline{x}]})]} \{\Theta \mid \Gamma, \overline{x_i} : \overline{\sigma_i} \vdash \mathrm{T}_i[\overline{s}, \overline{x_i}] \stackrel{?}{=} t_i : \tau_i\}^{i \in \{1,\dots,n\}} \qquad \textsf{(imitate}^{6}\textsf{)}$$

Note that SOAS does not specify the mechanism for name binding, α-equivalence, or capture-avoiding substitution. The implementation of the matching algorithm via Free Foil makes this choice automatic (using an efficient implementation [22] of the Barendregt's "no shadowing" convention [2] under the hood). Hence, for example the rule **(decompose)** does not consider the case when bound variables in matching scopes are named differently (i.e. $\mathsf{F}(x.t) \stackrel{?}{=} \mathsf{F}(y.s)$ is also handled by **(decompose)** with α-conversion being left as an implementation detail).

The proposed testing framework relies on matching for comparing against the reference solutions. Given two solutions θ_1 and θ_2, one can determine whether θ_1 is more general than θ_2 by checking if there exists a substitution η such that $\eta \circ \theta_1 = \theta_2$. Note that this approach only covers syntactic matching, and does not cover matching up to $\beta\eta$-equality or E-matching[7] in general.

3 Generalizing Huet's Preunification to SOAS

Huet's preunification algorithm [9] is a semi-decidable algorithm that only solves so called rigid-rigid and flex-rigid constraints, leaving flex-flex constraints unsolved. The original algorithm is given for the simply-typed lambda calculus, and the variation below generalizes it to work with an arbitrary object language specified in terms of SOAS and subject to some modest restrictions. In the generalized algorithm, a single step of the search performs the following:

1. **Normalize** both sides of all constraints in the constraint set;
2. **Decompose** all rigid-rigid constraints (possibly failing the current search branch);
3. Pick an arbitrary flex-rigid constraint and apply **imitation**, **projection**, and **introduction** rules to produce possible substitutions (if picking a constraint is not possible, a solution is found);
4. Apply each substitution in separate search branches and repeat from step 1.

The original algorithm assumes that terms are in normal form at all times; the generalization explicitly allows for non-normalized terms to appear temporarily after substitutions by including an explicit normalization step.

Note that a term may have subterms, and their value may change the outer term during normalization. In this work, such subterms are referred to as *heads* of the term. Then the classification is as follows:

- All variables are **rigid**;

[6] Since each imitation leads directly to another imitation or projection, resolving T_i, the implementation can avoid generating fresh metavariables T_i.

[7] A restricted version of E-unification for SOAS [12].

- All metavariable applications are **flexible**;
- Any user-defined construct is flexible iff any of its heads is flexible.

For example, a lambda abstraction is always rigid (since normalizations inside its body never affects the outer lambda), while a function application $f\ x$ is flexible iff the function f is flexible. A pair (x, y) is always rigid even if any of its components is not fully evaluated, because their evaluation may never replace the pair with another term.

$$(\Theta \mid \Gamma \vdash \mathrm{M}[\overline{s}] \stackrel{?}{=} \mathsf{F}(\overline{x.t}) : \tau) \xrightarrow{[\mathrm{M}[\overline{z}] \mapsto \mathsf{F}(\overline{x.u})]} \{\Theta \mid \Gamma, \overline{x_i} : \overline{\sigma_i} \vdash u_i \stackrel{?}{=} t_i : \tau_i\}^{i \in \{1,\dots,n\}} \quad \text{(imitate)}$$

$$\text{where } u_i = \begin{cases} t_i & \text{if } t_i \text{ is a head subterm} \\ \mathrm{T}_i[\overline{z}, \overline{x_i}] & \text{otherwise} \end{cases}$$

$$(\Theta \mid \Gamma \vdash \mathrm{M}[\overline{s}] \stackrel{?}{=} t : \tau) \xrightarrow{[\mathrm{M}[\overline{z}] \mapsto z_i]} \{\Theta \mid \Gamma \vdash s_i \stackrel{?}{=} t : \tau\} \quad (\text{project}_i)$$

$$(\Theta \mid \Gamma \vdash \mathsf{F}(\dots, \overline{x_i}.\mathrm{M}[\overline{s}], \dots) \stackrel{?}{=} t : \tau) \xrightarrow{[\mathrm{M}[\overline{z}] \mapsto \mathsf{G}(\overline{y}.\mathrm{T}[\overline{z}, \overline{y}])]} \{\Theta \mid \Gamma \vdash \mathsf{F}(\dots, \overline{x_i}.\mathsf{G}(\overline{y}.\mathrm{T}[\overline{s}, \overline{y}]), \dots) \stackrel{?}{=} t : \tau\} \quad (\text{introduce}_i)$$

$$\text{where } i\text{th subterm of } \mathsf{F} \text{ is a head}$$

The **(introduce$_i$)** rule is applicable to constraints where the flexible side is a complex term (not an immediate metavariable application), and produces substitutions for inner metavariables replaces that enable further normalization of the outer term. If the introduction rule is applied to a flex-rigid constraint, then the algorithm attempts to structurally decompose it in a separate search branch (without applying produced substitutions).

Overall, this rule allows one to encode equational unification. For example, consider $\mathrm{F}[]\ \mathsf{a} = \mathsf{a}$. Under β-reduction, the substitution $\mathrm{F}[] \mapsto \lambda x.x$ is a valid solution. To obtain such a solution, the introduction rule produces the substitution $\mathrm{F}[] \mapsto \lambda x.\mathrm{H}[x]$. Later steps of the algorithm will perform the substitution to obtain $(\lambda x.\mathrm{H}[x])\ \mathsf{a} = \mathsf{a}$, then normalize the constraint down to $\mathrm{H}[\mathsf{a}] = \mathsf{a}$, and finally produce the substitution $\mathrm{H}[x] \mapsto x$.

To justify further decomposition when the introduction rule is applied, consider the constraint $\mathrm{F}[]\ \mathsf{b} = \mathsf{f}\ \mathsf{b}$. Application of the introduction rule alone leads to the solution $\mathrm{F}[] \mapsto \lambda x.\mathsf{f}\ x$. However, the substitution $\mathrm{F}[] \mapsto f$ is also a valid solution. To obtain it, one has to decompose the constraint structurally. Yet this decomposition needs to be delayed to preserve the original constraint while the algorithm is solving other flex-rigid constraints.

Since the algorithm accounts for the semantics of the object language, a user of the algorithm needs to show normalization rules. The above rules should be possible to derive using normalization rules, but in practice it is easier to ask the user to implement the semantic parts of the algorithm.

The generalized algorithm is tested on an extension of a simply-typed lambda calculus with pairs and sum types. The extension proved to be quick and easy; the algorithm continued to produce correct solutions. A theoretical proof of soundness and completeness of the generalized algorithm is yet to be done.

4 Functions-as-Constructors Unification for SOAS

Functions-as-constructors Unification (FCU) algorithm [17] is an extension to Miller's Higher-order Pattern Unification. It has found significant application in many computational logic systems. Despite addressing a limited scope of unification problems, Pattern Unification is often enough for a system to be practically applicable.

The main distinctive trait of FCU algorithm is the way it treats arguments of metavariables. In the original pattern fragment, a metavariable M can only occur as $M[x_1, \dots, x_n]$, where x_i

are distinct variables. FCU extends the scope of the addressed problems by allowing these arguments to be the so-called "restricted terms". These restricted terms can be either variables or function symbols, as long as variables occupy all leaf positions.

Despite the extra generality, FCU remains a decidable unification procedure, built from six transformation rules and three restrictions:

- **Argument restriction**: every argument of a metavariable must be a restricted term;
- **Local restriction**: an argument of a metavariable must not be a subterm of another argument from that same metavariable;
- **Global restriction**: an argument of a metavariable must not be a *strict* subterm of another argument from another metavariable;

The algorithm operates using a set of six transformation rules that define the way terms are unified. Each step of the algorithm involves selecting and applying an appropriate rule, aiming to unify the given terms:

- **(0) Variable-variable** rule handles the trivial case where two variables are compared directly. If variables are identical, the constraint is solved immediately. Otherwise, the algorithm fails.
- **(1) Abstraction decomposition** handles abstraction terms, ensuring binders are properly unified and the scope structure is preserved through renaming and substitution.
- **(2) Rigid-rigid** decomposition rule handles terms whose heads are identical function symbols, structurally decomposing their arguments into separate unification problems.
- **(3) Flex-rigid** rule solves constraints involving a metavariable application (flexible) on one side and a function application on the other side, creating a substitution for the metavariable.
- **(4) Flex-flex with identical metavariables** resolves constraints where the same metavariable appears on both sides with different arguments lists of the same length, creating a substitution with fresh metavariable.
- **(5) Flex-flex with different metavariables** handles constraints where distinct metavariables appear on both sides, creating substitution for one of the metavariables.
- **Pruning**, applied before rules (3) and (5), removes redundant arguments from metavariable applications, simplifying the next unification steps.q

We present a generalization of FCU algorithm, implementing an abstraction over a generic term structure. Restrictions and transformation rules of the algorithm are explicitly encoded in the implementation as separate, modular functions with clear representation of the theoretical presentation. The user of the algorithm only needs to provide the functions for constructing applications and abstractions in the transformation rules' results.

Implemented generalization enhances the adaptability and expands the potential applications of FCU in various reasoning systems and computational logic frameworks. Utilizing Free Foil's scope-safe representation, it manages binders and scopes, ensuring correctness of the substitution operations. Proper testing of the implementation with the test suite and a theoretical proof of soundness and completeness are yet to be completed.

5 Future work

While Free Foil can be seen as an implementation of SOAS, it specifically relies on the Foil [18] which bears some resemblence with *permissive nominal* terms [7] and LS-nominal sets [27]. Further analysis and comparison is needed to compare and, possibly, relate the approaches, possibly leading to a nominal refinement of second-order abstract syntax.

Proposed algorithms and their implementations have been tested, but proper proofs for correctness and soundness is subject to future work. Matching is probably the easiest to start with, but the other two algorithms most likely require new proofs, as the proofs for original algorithms (for simply-typed lambda calculus) are highly dependent on the specific structure of λ-terms.

An application of the generalized algorithms to a toy dependent typechecker (e.g. Tog [19]) would be a good demonstration of their "language agnostic" design. It might also be good to compare such an implementation against existing ones.

The testing framework currently supports λ-calculus, extended with parametrized metavariables, and is not truly language agnostic. The inteface can be improved to support better comparison of HoU algorithms. There is also not many tests in our test suite to properly compare algorithms for correctness and performance.

Also, current implementations lack full documentation and examples.

References

[1] Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In Luke Ong, editor, *Typed Lambda Calculi and Applications*, pages 10–26, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[4] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[5] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[6] Gilles Dowek. Higher-order unification and matching. *Handbook of Automated Reasoning*, 2, 12 2001.

[7] Gilles Dowek, Murdoch J Gabbay, and Dominic P Mulligan. Permissive nominal terms and their unification: an infinite, co-infinite approach to nominal techniques. *Logic Journal of IGPL*, 18(6):769–822, 2010.

[8] Marcelo Fiore and Chung-Kil Hur. Second-Order Equational Logic (Extended Abstract). In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2010.

[9] G.P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[10] Víctor López Juan. *Practical Unification for Dependent Type Checking*. PhD thesis, Universidade Tecnica de Lisboa (Portugal), 2020.

[11] Edward Kmett. Rotten bananas, 03 2008. Accessed: 2024-11-07.

[12] Nikolai Kudasov. E-Unification for Second-Order Abstract Syntax. In Marco Gaboardi and Femke van Raamsdonk, editors, *8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023)*, volume 260 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:22, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[13] Nikolai Kudasov. Generalising Huet-style Projections in E-unification for Second-Order Abstract Syntax. In *UNIF 2023 - 37th International Workshop on Unification*, Rome, Italy, July 2023. Veena Ravishankar and Christophe Ringeissen.

[14] Nikolai Kudasov. Free Monads, Intrinsic Scoping, and Higher-Order Preunification, 2024. To appear in *TFP 2024*.

[15] Nikolai Kudasov. Free monads, intrinsic scoping, and higher-order preunification. In Jason Hemann and Stephen Chang, editors, *Trends in Functional Programming*, pages 22–54, Cham, 2025. Springer Nature Switzerland.

[16] Nikolai Kudasov, Renata Shakirova, Egor Shalagin, and Karina Tyulebaeva. Free foil: Generating efficient and scope-safe abstract syntax. In *2024 4th International Conference on Code Quality (ICCQ)*, pages 1–16, 2024.

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

[18] Dougal Maclaurin, Alexey Radul, and Adam Paszke. The foil: Capture-avoiding substitution with no sharp edges. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages*, IFL '22, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Francesco Mazzoli and Andreas Abel. Type checking through unification, 2016.

[20] Dale Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 09 1991.

[21] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

[22] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*, 12:393–434, 07 2002.

[23] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988.

[24] Robert Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks, Antibes*, volume 4, page 66. Citeseer, 1990.

[25] Wren Romano. `unification-fd`: Simple generic unification algorithms.

[26] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[27] Jan van Brügge, James McKinna, Andrei Popescu, and Dmitriy Traytel. Barendregt convenes with knaster and tarski: Strong rule induction for syntax with bindings. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.

[28] Philip Wadler. The expression problem, 11 1998. Accessed: 2024-11-07.