Unification Modulo Isomorphisms between Dependent Types for Type-Based Library Search

Satoshi Takimoto
Institute of Science Tokyo
Tokyo, Japan
takimoto@psg.comp.isct.ac.jp

Sosuke Moriguchi
Institute of Science Tokyo
Tokyo, Japan
chiguri@comp.isct.ac.jp

Takuo Watanabe
Institute of Science Tokyo
Tokyo, Japan
takuo@comp.isct.ac.jp

Abstract

Type-based library search allows developers to efficiently find reusable software components by their type signatures, as exemplified by tools like Hoogle. This capability is especially important in interactive theorem provers (ITPs), where reusing existing proofs can greatly accelerate development. Previous type-based library search tools for ITPs, such as SearchIsos and Loogle, support only a subset of desirable search flexibilities, including argument reordering, currying/uncurrying, generalisation, and the inclusion of extra premises. However, none can handle all these flexibilities simultaneously, resulting in missed relevant matches. In this work, we propose a type-based library search method based on equational unification modulo a set of type isomorphisms for dependent product/sum types, enabling all the desired search flexibilities. We present a semi-algorithm for this equational unification and provide a prototype implementation to demonstrate the feasibility of our approach.

CCS Concepts: • **Theory of computation → Type theory**; **Equational logic and rewriting**; **Automated reasoning**.

Keywords: unification, dependent types, type isomorphism, type-based library search

ACM Reference Format:
Satoshi Takimoto, Sosuke Moriguchi, and Takuo Watanabe. 2025. Unification Modulo Isomorphisms between Dependent Types for Type-Based Library Search. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3759538.3759651

1 Introduction

Effectively finding reusable software components in libraries is a fundamental challenge in software engineering. Enhancing discoverability directly contributes to code reuse and developer productivity.

TyDe '25, Singapore, Singapore

ACM ISBN 979-8-4007-2163-2/25/10
https://doi.org/10.1145/3759538.3759651

In 1989, Rittri proposed a powerful library search method based on types [20]. The motivation for this research was that searching by identifier is often insufficient because names are difficult to guess. For example, the recursor function for lists is given different names in various libraries and languages, such as "reduce", "foldr", or "itlist". In contrast, type-based library search allows users to find functions using types as queries, where types serve as an approximation of the specification. Since this seminal work, researchers have conducted extensive theoretical analyses and proposed various extensions to type-based search [6–8, 19, 21].

Type-based library search has been widely and successfully used in real-world. One notable example is Hoogle [17], a type-based library search engine for Haskell. According to Neil Mitchell, the original developer of Hoogle, between 1,000 and 2,500 searches are performed daily, indicating that Hoogle is a widely used and valuable tool for Haskell developers [18]. Type-based library search has also been introduced to other statically typed programming languages such as Scala [27] and Java [9].

Just as Haskell users benefit from Hoogle, users of interactive theorem provers (ITPs) frequently need to locate not only definitions but also lemmas and theorems by their types (i.e., propositions, via the Curry-Howard correspondence). Because proving theorems can be time-consuming, it is essential to effectively discover and reuse existing results. For ITPs, it is even more critical than in conventional programming languages, where proofs are rare. To meet this demand, several type-based library search tools have been developed, including SearchIsos [5], Loogle [3], and the search commands shipped with Rocq [25], Idris [11, 24], and Lambdapi [4, 15]. While each of these tools offers its own search flexibility and is highly beneficial, each misses some desirable search flexibilty. Enhancing the flexibility of type-based search engines can yield more comprehensive search results, thereby accelerating the process of finding relevant components or theorems.

Our work aims to develop a more flexible type-based search for ITPs, enabling users to discover results that previous systems might overlook. We propose using equational unification modulo a specific set of type isomorphisms as our solution. Our main contributions are:

- An enhanced theory of the type isomorphisms, improving upon the approach in [5]

- A semi-algorithm for unification modulo this theory
- A prototype implementation of the semi-algorithm

First of all, we present the problem framework and compares previous type-based library search tools in Section 2. Next, in Section 3, we introduce the dependent type theory that serves as the foundation for our work. We then discuss the type isomorphisms that are valid within the type theory and adopt those particularly beneficial for library search. After that, we build a semi-algorithm for equational unification modulo the chosen type isomorphisms in Section 4. Finally, we discuss our implementation and provide some examples that show the behavior of our algorithm in Section 5.

2 Problem Setting and Related Work

2.1 Beneficial Search Flexibilities

The simplest strategy for type-based library search is purely syntactic matching on types. However, this is often too rigid and may miss potentially relevant reusable components. More sophisticated strategies should have more flexible matching conditions. The followings are flexibilities considered beneficial in the context of ITPs.

Definitional Equality. Users may not be aware that the library includes a type alias for commutativity statement: $\mathrm{Commutative}\ A\ \mathrm{op} := (x\,y\colon A) \to \mathrm{op}\,x\,y \equiv \mathrm{op}\,y\,x$. If the search is up to definitional equality, specifically β-conversion and δ-conversion in this case, it can identify the following two types:

$$(m\,n\colon \mathbb{N}) \to m + n \equiv n + m$$
$$\mathrm{Commutative}\ \mathbb{N}\ \_+\_.$$

Reordering. Modulo reordering of function domains, pair components, and symmetry of identity types. Differences in the ordering of such components is superficial and users always want the search strategy to ignore it. For example, the following types should be identified. Note that the iterated function types in the argument position are considered for reordering in the first example.

$$B \to (B \to A \to B) \to \mathrm{List}\,A \to B$$
$$(A \to B \to B) \to B \to \mathrm{List}\,A \to B$$
$$(xs : \mathrm{Vec}\,A\,(m{+}n)) \to \Sigma ys{:}\mathrm{Vec}\,A\,m.\Sigma zs{:}\mathrm{Vec}\,A\,n.xs \equiv ys +\!\!+ zs$$
$$(xs : \mathrm{Vec}\,A\,(m{+}n)) \to \Sigma zs{:}\mathrm{Vec}\,A\,n.\Sigma ys{:}\mathrm{Vec}\,A\,m.xs \equiv ys +\!\!+ zs$$
$$(n\colon \mathbb{N}) \to n + 0 \equiv n$$
$$(n\colon \mathbb{N}) \to n \equiv n + 0$$

Currying and Uncurrying. Modulo the difference of curried function and uncurried function. While library components are often curried, this flexibility is helpful when functions take arguments of types defined using dependent pairs or records. In ITPs, the relation m divides n is usually defined as: $m \mid n := \Sigma q{:}\mathbb{N}.m \times q \equiv n$. However, a user may not be aware that this relation exists in the library and may search for a function involving two natural numbers where one divides the other:

$$(m\,n\,q : \mathbb{N}) \to m \times q \equiv n \to \cdots.$$

If the search is modulo currying/uncurrying (and definitional equality), it can identify

$$(m\,n : \mathbb{N}) \to m \mid n \to \cdots.$$

Generalisation. Search up to generalisation. For example, a user might query $(n\colon \mathbb{N}) \to n + 1 \equiv 1 + n$. If the search is up to generalisation, it can yield a more general result $(m\,n\colon \mathbb{N}) \to m + n \equiv n + m$.

More Premises and Arguments. Users do not always recognise enough premises required to derive a certain consequence. Even if they do, they may prefer not to include all the premises as it can be tedious to type. A user may query $(m\,n\colon \mathbb{N}) \to (m - n) + n \equiv m$. If the search has the flexibility, it can yield $(m\,n\colon \mathbb{N}) \to n \le m \to (m - n) + n \equiv m$.

2.2 Existing Type-based Library Search for ITPs

To support the demand for type-based library search, several tools have been developed for ITPs and dependently-typed programming languages. Notable examples include:

- SearchIsos, implemented in Coq previously [5]
- Rocq's Search command [25].
- Idris's search command [11, 24]
- Loogle, for Lean [3].
- Lambdapi's search command [4, 15]

Each of the search engines listed above has different objective and consequently supports a different subset of the flexibilities. Loogle and Lamdapi search through the whole standard libraries and such (global search) and Rocq and Idris search within scope or a few specified modules (local search). SearchIsos implements both. Local library search tools usually come with richer search flexibilities while global library search tools can handle a large set of definitions. Which flexibilities in the previous section are considered in these library search tools specifically? SearchIsos, Loogle and Lambdapi search modulo reordering domains in different ways, but these do not reorder iterated function types in argument position. Idris has an additional support for exploiting the symmetry of identity types. Rocq, Loogle, Lambdapi, and Idris support generalisation and the flexibility to introduce more premises (in different ways again). It is only SearchIsos that adopts the currying/uncurrying flexibility. None of them considers δ-conversion.

2.3 Our Work

Our work aims to support all the flexibilities listed above. We achieve this by employing equational unification modulo definitional equality ($\beta\eta\delta$-conversion) and a chosen collection of type isomorphisms. Reordering and currying/uncurrying are handled by finding unifiers modulo these type isomorphisms.

We support generalisation and the extra premises flexibility through the use of metavariables. In particular, the latter flexibility is realised by augmenting the query type with an argument of unknown type, represented by a metavariable. For example, given the query $(m\,n\colon \mathbb{N}) \to (m - n) + n \equiv m$, we synthesise $(m\,n\colon \mathbb{N}) \to M[m, n] \to (m - n) + n \equiv m$, where M is a metavariable that may depend on m and n.

Our approach currently targets local search since we consider δ-conversion; indexing all definitions from all available libraries would be prohibitively expensive. Omitting δ-conversion yields a more scalable unification algorithm, but it is still too costly to compare with each definition in libraries for global search. This can probably be addressed by term indexing techniques. For example, feature vector indexing [23] can be employed to filter out candidates whose type definitely cannot be unified with, thus reducing the number of unification calls. Allain et al. reported that their type-based library search tool for OCaml that implements the indexing can quickly execute queries over the entire OCaml environment [2]. Global search remains an important goal we plan to address in future work.

3 Types and Isomorphisms

3.1 Types

The type theory used in this paper is a version of Martin-Löf type theory with identity types and constant definitions. We use x, y, z for bound variables and c, d for constants.

$$\begin{aligned} t, u, p, A, B, C ::= {} & \mathrm{Type} \mid \Pi x{:}A.B \mid \Sigma x{:}A.B \mid \mathrm{Unit} \\ & \mid \mathrm{Id}_A(t, u) \mid x \mid c \mid \lambda x.t \mid t\,u \\ & \mid (t, u) \mid \pi_1(t) \mid \pi_2(t) \mid \mathrm{tt} \\ & \mid \mathrm{refl}_A(t) \mid \mathrm{J}(t_1, t_2, p, u) \end{aligned}$$

Terms are identified modulo α-conversion. We also denote $\Pi x{:}A.B$ and $\Sigma x{:}A.B$, respectively $A \to B$ and $A \times B$ when $x \notin \mathrm{FV}(B)$, where $\mathrm{FV}(B)$ is the set of free variables in B. A signature $\mathcal{S}$ is a collection of constant definitions, which take the form $c\colon A := t$. The typing rules are summarised in Figure 1 in which we use type-in-type for simplicity of presentation. The reduction rules are defined as usual from the following one-step rules.

$$\begin{aligned} (\lambda x.t)\,u &\rightsquigarrow t[x \mapsto u] && (\beta_\lambda) \\ \pi_1((t, u)) &\rightsquigarrow t && (\beta_1) \\ \pi_2((t, u)) &\rightsquigarrow u && (\beta_2) \\ c &\rightsquigarrow t \quad \text{if } c\colon A := t \in \mathcal{S} && (\delta) \end{aligned}$$

The judgment $t =_\eta u$ compares t and u modulo η, i.e., modulo $(\lambda x.t\,x) = t$ if $x \notin \mathrm{FV}(t)$, $t = (\pi_1(t), \pi_2(t))$, and $t = \mathrm{tt}$ if $t\colon \mathrm{Unit}$. Two terms t and u are said definitionally equal or $\beta\eta\delta$-convertible, written $t \simeq u$, if only if they can be reduced to t' and u' respectively and $t' =_\eta u'$.

3.2 Isomorphisms

As in the previous work on type-based library search including SearchIsos [5, 7, 8, 19–21], we adopt definitional isomorphism as the foundational equivalence relation for type-based library search in order to capture the reordering and currying/uncurrying flexibility. Informally, two types are definitionally isomorphic if there exist mutually inverse functions converting between them. This is formally defined as follows.

Definition 1. (Definitional isomorphisms) Two types A and B are (definitionally) isomorphic, written $A \cong B$, if and only if there exist the terms $f\colon A \to B$ and $f^{-1}\colon B \to A$ such that $f^{-1} \circ f \simeq \mathrm{id}_A$ and $f \circ f^{-1} \simeq \mathrm{id}_B$, where id_A is the identity function on A and $\circ$ is function composition. We refer to the pair of f and f^{-1} as a definitional isomorphism between A and B, and each f and f^{-1} as conversion functions between A and B.

Isomorphisms are particularly well-suited for type-based search because they are accompanied by explicit terms that translate between types within the language. This not only enables the identification of matching software components but also allows for the automatic generation of conversion functions to adapt matched components to the user's query. There are other forms of isomorphisms, such as propositional isomorphisms, which we intend to explore in future work.

Now we can formally state the isomorphisms we consider for type-based search. Specifically, we adopt the following isomorphisms (with the standard conversion functions), which we will take as equations over types and use for equational unification. This set of isomorphisms is similar to that in SearchIsos [5], but we have excluded two isomorphisms and added reordering for dependent products and identity types.

$$A \cong B \ \text{ if } A \simeq B \tag{1}$$
$$A \times B \cong B \times A \tag{2}$$
$$\Sigma x{:}(\Sigma y{:}A.B).C \cong \Sigma x{:}A.\Sigma y{:}B[y \mapsto x].C[x \mapsto (x, y)] \tag{3}$$
$$\Pi x{:}(\Sigma y{:}A.B).C \cong \Pi x{:}A.\Pi y{:}B[y \mapsto x].C[x \mapsto (x, y)] \tag{4}$$
$$\Pi x{:}A.\Pi y{:}B.C \cong \Pi y{:}B.\Pi x{:}A.C \ \text{ if } x \notin \mathrm{FV}(B) \wedge y \notin \mathrm{FV}(A) \tag{5}$$
$$A \times \mathrm{Unit} \cong A \tag{6}$$
$$\Sigma x{:}\mathrm{Unit}.A \cong A[x \mapsto \mathrm{tt}] \tag{7}$$
$$\Pi x{:}\mathrm{Unit}.A(x) \cong A[x \mapsto \mathrm{tt}] \tag{8}$$
$$\mathrm{Id}_A(t, u) \cong \mathrm{Id}_A(u, t) \tag{9}$$

Note that the reordering for dependent products can be derived from the corresponding isomorphism for dependent sums together with currying. To justify that all of the isomorphisms conform to Definition 1, one additional assumption is required. Specifically, for the symmetry of identity types (Equation (9)), the swap operation $\mathrm{Id}_A(t, u) \to \mathrm{Id}_A(u, t)$ must be definitionally involutive. This property holds, for example,

$$\frac{}{\Gamma \vdash \mathrm{Type} : \mathrm{Type}} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{c : A := t \in \mathcal{S}}{\Gamma \vdash c : A}$$

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma, x : A \vdash B : \mathrm{Type}}{\Gamma \vdash \Pi x{:}A.B : \mathrm{Type}} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \Pi x{:}A.B} \qquad \frac{\Gamma \vdash t : \Pi x{:}A.B \quad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B[x \mapsto u]}$$

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma, x : A \vdash B : \mathrm{Type}}{\Gamma \vdash \Sigma x{:}A.B : \mathrm{Type}} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B[x \mapsto t]}{\Gamma \vdash (t, u) : \Sigma x{:}A.B} \qquad \frac{\Gamma \vdash p : \Sigma x{:}A.B}{\Gamma \vdash \pi_1(p) : A}$$

$$\frac{\Gamma \vdash p : \Sigma x{:}A.B}{\Gamma \vdash \pi_2(p) : B[x \mapsto \pi_1(p)]} \qquad \frac{}{\Gamma \vdash \mathrm{Unit} : \mathrm{Type}} \qquad \frac{}{\Gamma \vdash \mathrm{tt} : \mathrm{Unit}} \qquad \frac{\Gamma \vdash A : \mathrm{Type}}{\Gamma, t : A, u : A \vdash \mathrm{Id}_A(t, u) : \mathrm{Type}}$$

$$\frac{\Gamma \vdash A : \mathrm{Type}}{\Gamma, t : A \vdash \mathrm{refl}_A(t) : \mathrm{Id}_A(t, t)} \qquad \frac{\Gamma, t_1 : A, t_2 : A, p : \mathrm{Id}_A(t_1, t_2) \vdash C : \mathrm{Type} \quad \Gamma, t : A \vdash u : C[t_1 \mapsto t, t_2 \mapsto t, p \mapsto \mathrm{refl}_A(t)]}{\Gamma, t_1 : A, t_2 : A, p : \mathrm{Id}_A(t_1, t_2) \vdash \mathrm{J}(t_1, t_2, p, u) : C}$$

Figure 1. Typing rules

for the Path type in Cubical Agda [26], or in settings where uniqueness of identity proofs holds definitionally as in Lean. However, it does not generally hold for the conventional identity type defined inductively with the canonical reflexivity constructor. Since this reordering flexibility is valuable for type-based search, we assume that identity types are strictly involutive. The two excluded isomorphisms are as follows:

$$A \to \mathrm{Unit} \cong \mathrm{Unit}$$
$$\Pi x{:}A.\Sigma y{:}B(x).C(x, y) \cong \Sigma y{:}(\Pi x{:}A.B(x)).\Pi x{:}A.C(x, y\ x).$$

The first isomorphism relates functions from A to Unit with Unit, while the second relates dependent functions returning a dependent pair to a dependent pair of functions returning each component. We exclude the isomorphism because it has limited practical relevance for library search. The second isomorphism is excluded due to a technical restriction, which will be explained in Section 4.4.

3.3 Theory

As pointed out in [5], with dependencies, we cannot build a contextual theory taking the above isomorphisms as equations. For example, given an isomorphism $A \cong B$ and an A-indexed type C, and let $f : A \to B, f^{-1} : B \to A$ be conversion functions of the isomorphism. It makes no sense to ask if $\Pi x{:}A.C\,x \cong \Pi x{:}B.C\,x$ because the right hand side is ill-typed. We should instead ask if $\Pi x{:}A.C\,x \cong \Pi x{:}B.C\,(f^{-1}\,x)$.

There is another interesting subtlety. The conversion functions between the two types are as follows:

$$g : (\Pi x{:}A.C(x)) \to (\Pi x{:}B.C(f^{-1}\,x)) = \lambda h.\lambda x.h(f^{-1}\,x)$$
$$g^{-1} : (\Pi x{:}B.C(f^{-1}\,x)) \to (\Pi x{:}A.C(f^{-1}(f\,x))) = \lambda h.\lambda y.f\,x$$

From the definition of isomorphism, $C(f^{-1}(f\,x))$ is equal to $C\,x$, and thus $g^{-1} \circ g : (\Pi x{:}A.C(x)) \to (\Pi x{:}A.C(x))$. However, if we choose to opt out of certain η-rules or do not assume that identity types are definitionally involutive, the type $C(f^{-1}(f\,x))$ may not reduce further. In this case, $g^{-1} \circ g$ does not have the correct type, i.e., that of the identity function. The issue is that the conversion functions can appear within types. For this reason, SearchIsos does not allow isomorphisms at the domain of dependent product/sum types at all [5]. In our work, we relax the restriction slightly and allow isomorphisms at the domain of non-dependent function/product types. In this case the isomorphisms for the domain can be applied safely as conversion functions do not appear in type. This relaxation enables, for example, the reordering iterated function types in argument position, increasing the chances of identifying reusable components in type-based search. The resulting theory, Th, is shown in Figure 2. The distinction between the dependent case and the non-dependent case is reflected in the rules $\mathrm{Th}_{\Pi L}$, $\mathrm{Th}_{\to L}$, $\mathrm{Th}_{\Sigma L}$, and $\mathrm{Th}_{\times L}$. Hereafter, we refer to the restriction of disallowing isomorphisms at the domain as the domain restriction.

4 Pre-unification Modulo Isomorphisms

Here, we present a semi-algorithm for pre-unification modulo the theory described in Section 3. It is loosely based on E-unification for second-order abstract syntax [13, 14]. The algorithm is general in that can handle arbitrary second-order equational theory including λ-calculus.

We only implement pre-unification and currently leave flex-flex constraints unsolved. This is because going through candidate solutions for flex-flex constraints would be quite expensive, which leads to a slow response to the user of search tool built upon this algorithm. The practical impact of solving flex-flex constraints remains an open question which we plan to address.

Our pre-unification procedure operates in an untyped setting. The reason is that the query term is arbitrary and can even be ill-typed. While this reduces the guarantees of well-typedness during unification, type correctness of resulting unifiers can be checked by post-validation. Type information from libraries can be taken into account during unification for benefits. For example, leveraging type information can

$$\frac{A \simeq B}{A =_{\mathrm{Th}} B}\ \mathrm{Th}_{\beta\eta\delta} \qquad \frac{}{\Sigma x{:}\mathrm{Unit}.A =_{\mathrm{Th}} A[x \mapsto \mathrm{tt}]}\ \mathrm{Th}_{\Sigma UL} \qquad \frac{}{A \times \mathrm{Unit} =_{\mathrm{Th}} A}\ \mathrm{Th}_{\Sigma UR} \qquad \frac{}{\Pi x{:}\mathrm{Unit}.A =_{\mathrm{Th}} A[x \mapsto \mathrm{tt}]}\ \mathrm{Th}_{\Pi UL}$$

$$\frac{x \notin \mathrm{FV}(A) \quad x \notin \mathrm{FV}(B)}{\Sigma x{:}A.B =_{\mathrm{Th}} \Sigma x{:}B.A}\ \mathrm{Th}_{\Sigma\mathrm{Comm}} \qquad \frac{x \notin \mathrm{FV}(A) \quad y \notin \mathrm{FV}(B)}{\Pi x{:}A.\Pi y{:}B.C =_{\mathrm{Th}} \Pi y{:}B.\Pi x{:}A.C}\ \mathrm{Th}_{\Pi\mathrm{Swap}} \qquad \frac{}{\mathrm{Id}_A(t,u) =_{\mathrm{Th}} \mathrm{Id}_A(u,t)}\ \mathrm{Th}_{\mathrm{IdSwap}}$$

$$\frac{}{\Pi x{:}(\Sigma y{:}A.B).C =_{\mathrm{Th}} \Pi x{:}A.\Pi y{:}B[y{\mapsto}x].C[x{\mapsto}(x,y)]}\ \mathrm{Th}_{\mathrm{Curry}} \qquad \frac{}{\Sigma x{:}(\Sigma y{:}A.B).C =_{\mathrm{Th}} \Sigma x{:}A.\Sigma y{:}B[y{\mapsto}x].C[x{\mapsto}(x,y)]}\ \mathrm{Th}_{\Sigma\mathrm{Assoc}}$$

$$\frac{B =_{\mathrm{Th}} B'}{(\Pi x{:}A.B) =_{\mathrm{Th}} (\Pi x{:}A.B')}\ \mathrm{Th}_{\Pi R} \qquad \frac{x \in \mathrm{FV}(B) \quad A \simeq A'}{(\Pi x{:}A.B) =_{\mathrm{Th}} (\Pi x{:}A'.B)}\ \mathrm{Th}_{\Pi L} \qquad \frac{A =_{\mathrm{Th}} A'}{(A \to B) =_{\mathrm{Th}} (A' \to B)}\ \mathrm{Th}_{\to L}$$

$$\frac{B =_{\mathrm{Th}} B'}{(\Sigma x{:}A.B) =_{\mathrm{Th}} (\Sigma x{:}A.B')}\ \mathrm{Th}_{\Sigma R} \qquad \frac{x \in \mathrm{FV}(B) \quad A \simeq A'}{(\Sigma x{:}A.B) =_{\mathrm{Th}} (\Sigma x{:}A'.B)}\ \mathrm{Th}_{\Sigma L} \qquad \frac{A =_{\mathrm{Th}} A'}{(A \times B) =_{\mathrm{Th}} (A' \times B)}\ \mathrm{Th}_{\times L}$$

$$\frac{A =_{\mathrm{Th}} B}{B =_{\mathrm{Th}} A}\ \mathrm{Th}_{\mathrm{Sym}} \qquad \frac{A =_{\mathrm{Th}} B \quad B =_{\mathrm{Th}} C}{A =_{\mathrm{Th}} C}\ \mathrm{Th}_{\mathrm{Trans}}$$

Figure 2. Theory Th

help prune impossible branches early when enumerating candidates for metavariables. We leave integrating such type-driven guidance and keeping well-typedness for future work.

4.1 Second-order Abstract Syntax and Parametrised Metavariables

For unification, we need to add metavariables to our syntax. We follow the framework of second-order abstract syntax (SOAS), simply-typed syntax with variable binding and parametrised metavariables [10]. Higher-order unification relies on function application and metavariables are applied to terms in context to represent dependencies. On the other hand, parametrised metavariables can keep dependencies in a more direct way. One can think a parametrised metavariables similar to flex terms in higher-order unification.

We use M for metavariables and write $M[t_1, \cdots, t_n]$ for meta-application. The type signature of a parametrised metavariable is like $M : [\star, \cdots, \star]\star$. Here $\star$ means the "any" type (we are working in an untyped setting). For example, if M_1 takes two parameters then $M_1 : [\star, \star]\star$, and if M_2 no parameter, $M_2 : []\star$.

4.2 Pre-unification Semi-algorithm

We present the pre-unification algorithm as transition rules on a set of constraints. A constraint is a judgement of form $\Theta \mid \Gamma \vdash t \stackrel{?}{\cong} u$ or $\Theta \mid \Gamma \vdash t \stackrel{?}{=} u$ where Θ is the context of (parametrised) metavariables, Γ is the context of bound variables, and t and u are terms in context that we would like to unify. $\stackrel{?}{\cong}$ and $\stackrel{?}{=}$ mean that the constraint is to be solved up to the type isomorphisms and definitional equality respectively. We start the unification with $\stackrel{?}{\cong}$ and switch to $\stackrel{?}{=}$ when the domain restriction of the theory applies[1]. We notate a substitution of metavariables as follows: $[M_1[x_1, \cdots, x_{n_1}] \mapsto t_1, \cdots, M_m[x_1, \cdots, x_{n_m}] \mapsto t_m]$, where each t_i may use parameters $x_1, \cdots, x_{n_i}$. We write each transition rule of the unification procedure in the following form $\Theta \mid \Gamma \vdash t \stackrel{?}{\cong} u \overset{\theta}{\longmapsto} \{\Xi \mid \Gamma_i \vdash t_i \stackrel{?}{\cong} u_i\}$, where Ξ is a new metavariable context, θ is a metavariable substitution and $\{\Xi \mid \Gamma_i \vdash t_i \stackrel{?}{\cong} u_i\}$ is a new set of constraints.

The unification problem consists of solving a set of constraints via the following process:

1. Fully normalise both sides of each constraint using the **normalise** rule (Figure 5) before applying other transition rules.
2. Simplify each constraint by decomposing composite terms (the **decompose** and **permute-decompose** rule, Figures 3 and 6) and guessing the structure of metavariables (the **guess** rule, Figure 8). In case multiple simplifications can be applied, choose one non-deterministically.
3. Attempt to solve flex-rigid constraints that cannot be simplified further.
4. If a constraint can be solved and a substitution for a metavariables is determined, propagate the solution throughout the remaining constraints. If no solution can be found, backtrack.
5. Repeat until only flex-flex constraints remain.

We describe the details in the following subsections. We omit other transition rules including decomposition of lambda abstractions and so on as they are straightforward.

4.3 Decomposition

During unification, each constraint is gradually decomposed into a set of simpler constraints using the **decompose** rule (Figure 3). Because of the domain restriction, whether a constraint should be solved modulo isomorphism or definitional

[1] We can start with $\stackrel{?}{=}$ to turn off search modulo the type isomorphisms.

equality must be set correctly at decomposition. If a constraint is in "definitional mode", there is no chance returning to the other mode and its derived constraints should always be solved up to defintional equality. On the other hand, the "up-to-isomorphism mode" is always propagated to the codomain of dependent product/sum types, but not always the case for the domain part. For example, a constraint on dependent product types will be decomposed as follows:

$$\Theta \mid \Gamma \vdash (\Pi x{:}A_1.A_2) \stackrel{?}{\cong} (\Pi x{:}B_1.B_2)$$
$$\longmapsto \{\Theta \mid \Gamma \vdash A_1 \stackrel{?}{=} B_1, \Theta \mid \Gamma, x \vdash A_2 \stackrel{?}{\cong} B_2\}.$$

The domain restriction requires that the constraint of the domains is solved up to definitional equality. However, if either side is actually a non-dependent function type, we can solve the constraint for the domain up to the isomorphisms.

$$\Theta \mid \Gamma \vdash (A_1 \to A_2) \stackrel{?}{\cong} (\Pi x{:}B_1.B_2)$$
$$\longmapsto \{\Theta \mid \Gamma \vdash A_1 \stackrel{?}{\cong} B_1, \Theta \mid \Gamma, x \vdash A_2 \stackrel{?}{\cong} B_2\}$$

In other words, different decomposition rules are applied depending on dependencies for constraints on dependent product/sum types in order to take isomorphisms at the domain position into account. Note that, although less precise, it remains sound to approximate that the domains of both sides are relevant, always choosing the former rule.

4.4 Normalisation

To unify two terms modulo isomorphisms, it is necessary to mutate terms in constraints using the theory. We orient a fragment of the equations of Th to obtain a rewrite system R. Both sides of each constraint are fully normalised with the use of R, before applying other transition rules (the **normalise** rule, Figure 5). We will explain how we take the remaining equations into account during unification by the **permute-decompose** rule in Section 4.5. The extracted rewrite system R is presented in Figure 4. This is roughly the same as the one used by SearchIsos [5] except that it includes rewrite rules for the two isomorphisms we do not adopt. Note that normalisation by R may not terminate since we are working in an untyped setting and thus not all terms are normalisable. We can handle this by using timeouts in practice.

Having presented the rewrite system R, we now explain why we excluded the following isomorphism:

$$\Pi x{:}A.\Sigma y{:}B(x).C(x,y) \cong \Sigma y{:}(\Pi x{:}A.B(x)).\Pi x{:}A.C(x,y\,x).$$

Consider the following two types:

$$\Pi x{:}A.\Pi y{:}A'.\Sigma z{:}B(x,y).C(z)$$
$$\Pi y{:}A'.\Pi x{:}A.\Sigma z{:}B(x,y).C(z).$$

These types differ only in the order of arguments, which our algorithm can handle using the **permute-decompose** rule. If we were to adopt the isomorphism and include it in the rewrite system R, orienting it from left to right, these types would normalise as follows:

$$\Sigma z{:}(\Pi x{:}A.\Pi y{:}A'.B(x,y)).\Pi x{:}A.\Pi y{:}A'.C(z\,x\,y)$$
$$\Sigma z{:}(\Pi y{:}A'.\Pi x{:}A.B(x,y)).\Pi y{:}A'.\Pi x{:}A.C(z\,y\,x).$$

At this point, the domain restriction prevents us from considering the isomorphism between these two types. As this example illustrates, the combination of domain restriction and the excluded isomorphism can sometimes block domain reordering. Since domain reordering is more beneficial for library search, we choose to prioritize it and exclude the isomorphism. A less general isomorphism could be adopted, restricting the return type to non-dependent products, but we do not currently pursue this approach.

4.5 Permutation

The non-orientable isomorphisms, i.e. reordering domains/-components of dependent product/sum types and symmetry of of identity types, are conceptually taken into account by non-deterministically permuting these components during unification. However, unrestricted application of permutations can cause exponential blow-up in the search space and infinite loops where only permutations are repeatedly applied without progress. To mitigate this, we restrict such permutation to happen only at decomposition (the **permute-decompose** rule, Figure 6). For example, when both sides of a constraint are identity types, we not only apply the standard decomposition but also try swapping the sides:

$$\Theta \mid \Gamma \vdash \mathrm{Id}_A(t_1,t_2) \stackrel{?}{\cong} \mathrm{Id}_B(u_1,u_2)$$
$$\longmapsto \{\Theta \mid \Gamma \vdash A \stackrel{?}{=} B, \Theta \mid \Gamma \vdash t_2 \stackrel{?}{=} u_1, \Theta \mid \Gamma \vdash t_1 \stackrel{?}{=} u_2\}$$

Similarly, for iterated dependent product/sum types, we permute domains to bring one to the front, respecting dependencies, then proceed with decomposition. For instance,

$$\Theta \mid \Gamma \vdash \Pi B{:}\mathrm{Type}.A{\to}B{\to}A \stackrel{?}{\cong} A{\to}\Pi B{:}\mathrm{Type}.B{\to}A \longmapsto$$
$$\{\Theta \mid \Gamma \vdash A \stackrel{?}{\cong} A, \Theta \mid \Gamma \vdash \Pi B{:}\mathrm{Type}.B{\to}A \stackrel{?}{\cong} \Pi B{:}\mathrm{Type}.B{\to}A\}.$$

Permutation is not performed when either side is headed by a metavariable, like $\Theta \mid \Gamma \vdash \Pi x{:}A.B \stackrel{?}{\cong} M[\overrightarrow{t}]$, in order to avoid excessive branching, trading some precision for performance.

To enable permutation in the presence of dependencies, we sometimes need to prune certain metavariable parameters that block valid reordering for iterated dependent product and sum types. Consider $\Pi A{:}\mathrm{Type}.M[x] \to A$. We cannot move $M[A]$ before $\Pi A : \mathrm{Type}$ because the metavariable M depends on A. By pruning the parameter, i.e., $[M[x] \mapsto ?N[]]$ where N is a fresh metavariable, we obtain $\Pi A{:}\mathrm{Type}.N[]{\to}A$, whose domains can now be reordered to $N[] \to \Pi A{:}\mathrm{Type}.A$.

We formalise pruning with the auxiliary judgment $t \,/\, \Delta \Rightarrow P$ (Figure 7). Here, t is a term that may contain metavariables, Δ is a telescope (a list of variables with types) whose

$$\Theta \mid \Gamma \vdash \mathrm{Type} \stackrel{?}{=} \mathrm{Type} \longmapsto \emptyset$$
$$\Theta \mid \Gamma \vdash (\Pi x{:}A_1.A_2) \stackrel{?}{=} (\Pi x{:}B_1.B_2) \longmapsto \{\Theta \mid \Gamma \vdash A_1 \stackrel{?}{=} B_1, \Theta \mid \Gamma, x \vdash A_2 \stackrel{?}{=} B_2\}$$
$$\Theta \mid \Gamma \vdash (\Sigma x{:}A_1.A_2) \stackrel{?}{=} (\Sigma x{:}B_1.B_2) \longmapsto \{\Theta \mid \Gamma \vdash A_1 \stackrel{?}{=} B_1, \Theta \mid \Gamma, x \vdash A_2 \stackrel{?}{=} B_2\}$$
$$\Theta \mid \Gamma \vdash \mathrm{Unit} \stackrel{?}{=} \mathrm{Unit} \longmapsto \emptyset$$
$$\Theta \mid \Gamma \vdash \mathrm{Id}_A(t_1, t_2) \stackrel{?}{=} \mathrm{Id}_B(u_1, u_2) \longmapsto \{\Theta \mid \Gamma \vdash A \stackrel{?}{=} B, \Theta \mid \Gamma \vdash t_1 \stackrel{?}{=} u_1, \Theta \mid \Gamma \vdash t_2 \stackrel{?}{=} t_2\}$$

$$\Theta \mid \Gamma \vdash \mathrm{Type} \stackrel{?}{\cong} \mathrm{Type} \longmapsto \emptyset$$
$$\Theta \mid \Gamma \vdash (\Pi x{:}A_1.A_2) \stackrel{?}{\cong} (\Pi x{:}B_1.B_2) \longmapsto \{\Theta \mid \Gamma \vdash A_1 \stackrel{?}{\cong} B_1, \Theta \mid \Gamma, x \vdash A_2 \stackrel{?}{\cong} B_2\} \quad \text{if } x \notin \mathrm{FV}(A_2) \vee x \notin \mathrm{FV}(B_2)$$
$$\Theta \mid \Gamma \vdash (\Pi x{:}A_1.A_2) \stackrel{?}{\cong} (\Pi x{:}B_1.B_2) \longmapsto \{\Theta \mid \Gamma \vdash A_1 \stackrel{?}{=} B_1, \Theta \mid \Gamma, x \vdash A_2 \stackrel{?}{\cong} B_2\} \quad \text{if } x \in \mathrm{FV}(A_2) \wedge x \in \mathrm{FV}(B_2)$$
$$\Theta \mid \Gamma \vdash (\Sigma x{:}A_1.A_2) \stackrel{?}{\cong} (\Sigma x{:}B_1.B_2) \longmapsto \{\Theta \mid \Gamma \vdash A_1 \stackrel{?}{\cong} B_1, \Theta \mid \Gamma, x \vdash A_2 \stackrel{?}{\cong} B_2\} \quad \text{if } x \notin \mathrm{FV}(A_2) \vee x \notin \mathrm{FV}(B_2)$$
$$\Theta \mid \Gamma \vdash (\Sigma x{:}A_1.A_2) \stackrel{?}{\cong} (\Sigma x{:}B_1.B_2) \longmapsto \{\Theta \mid \Gamma \vdash A_1 \stackrel{?}{=} B_1, \Theta \mid \Gamma, x \vdash A_2 \stackrel{?}{\cong} B_2\} \quad \text{if } x \in \mathrm{FV}(A_2) \wedge x \in \mathrm{FV}(B_2)$$
$$\Theta \mid \Gamma \vdash \mathrm{Unit} \stackrel{?}{\cong} \mathrm{Unit} \longmapsto \emptyset$$
$$\Theta \mid \Gamma \vdash \mathrm{Id}_A(t_1, t_2) \stackrel{?}{\cong} \mathrm{Id}_B(u_1, u_2) \longmapsto \{\Theta \mid \Gamma \vdash A \stackrel{?}{=} B, \Theta \mid \Gamma \vdash t_1 \stackrel{?}{=} u_1, \Theta \mid \Gamma \vdash t_2 \stackrel{?}{=} u_2\}$$

Figure 3. The **decompose** rules. The rules for the other syntactic constructs are omitted.

$$\frac{A \rightsquigarrow B}{A \rightsquigarrow_{\mathrm{R}} B}\,\mathrm{R}_{\beta\delta} \qquad \frac{}{\Sigma x{:}\mathrm{Unit}.A \rightsquigarrow_{\mathrm{R}} A[x \mapsto \mathrm{tt}]}\,\mathrm{R}_{\Sigma UL} \qquad \frac{}{A \times \mathrm{Unit} \rightsquigarrow_{\mathrm{R}} A}\,\mathrm{R}_{\Sigma UR} \qquad \frac{}{\Pi x{:}\mathrm{Unit}.A \rightsquigarrow_{\mathrm{R}} A[x \mapsto \mathrm{tt}]}\,\mathrm{R}_{\Pi UL}$$

$$\frac{}{\Pi x{:}(\Sigma y{:}A.B).C \rightsquigarrow_{\mathrm{R}} \Pi x{:}A.\Pi y{:}B[y{\mapsto}x].C[x{\mapsto}(x,y)]}\,\mathrm{R}_{\mathrm{Curry}} \qquad \frac{}{\Sigma x{:}(\Sigma y{:}A.B).C \rightsquigarrow_{\mathrm{R}} \Sigma x{:}A.\Sigma y{:}B[y{\mapsto}x].C[x{\mapsto}(x,y)]}\,\mathrm{R}_{\Sigma\mathrm{Assoc}}$$

$$\frac{B \rightsquigarrow_{\mathrm{R}} B'}{(\Pi x{:}A.B) \rightsquigarrow_{\mathrm{R}} (\Pi x{:}A.B')}\,\mathrm{R}_{\Pi R} \qquad \frac{x \in \mathrm{FV}(B) \quad A \rightsquigarrow A'}{(\Pi x{:}A.B) \rightsquigarrow_{\mathrm{R}} (\Pi x{:}A'.B)}\,\mathrm{R}_{\Pi L} \qquad \frac{A \rightsquigarrow_{\mathrm{R}} A'}{(A \to B) \rightsquigarrow_{\mathrm{R}} (A' \to B)}\,\mathrm{R}_{\to L}$$

$$\frac{B \rightsquigarrow_{\mathrm{R}} B'}{(\Sigma x{:}A.B) \rightsquigarrow_{\mathrm{R}} (\Sigma x{:}A.B')}\,\mathrm{R}_{\Sigma R} \qquad \frac{x \in \mathrm{FV}(B) \quad A \rightsquigarrow A'}{(\Sigma x{:}A.B) \rightsquigarrow_{\mathrm{R}} (\Sigma x{:}A'.B)}\,\mathrm{R}_{\Sigma L} \qquad \frac{A \rightsquigarrow_{\mathrm{R}} A'}{(A \times B) \rightsquigarrow_{\mathrm{R}} (A' \times B)}\,\mathrm{R}_{\times L}$$

Figure 4. Rewrite system R

$$\Theta \mid \Gamma \vdash A \stackrel{?}{=} B \longmapsto \{\Theta \mid \Gamma \vdash A' \stackrel{?}{=} B\} \quad \text{if } A \rightsquigarrow^* A'$$
$$\Theta \mid \Gamma \vdash A \stackrel{?}{=} B \longmapsto \{\Theta \mid \Gamma \vdash A \stackrel{?}{=} B'\} \quad \text{if } B \rightsquigarrow^* B'$$
$$\Theta \mid \Gamma \vdash A \stackrel{?}{\cong} B \longmapsto \{\Theta \mid \Gamma \vdash A' \stackrel{?}{\cong} B\} \quad \text{if } A \rightsquigarrow^*_{\mathrm{R}} A'$$
$$\Theta \mid \Gamma \vdash A \stackrel{?}{\cong} B \longmapsto \{\Theta \mid \Gamma \vdash A \stackrel{?}{\cong} B'\} \quad \text{if } B \rightsquigarrow^*_{\mathrm{R}} B'$$

Figure 5. The **normalise** rules.

dependencies we would like to eliminate by pruning, and P is the set of metavariable parameters that need to be pruned. We write (M, i) to mean the i-th parameter of the metavariable M. If any variable in Δ appears outside of metavariable arguments (i.e., occurs rigidly), we cannot eliminate the dependency by pruning, and thus no derivation exists in such cases. Examples:

- $M[x]$ / $x : A \Rightarrow \{(M, 0)\}$
- $M[x]\ N[x]$ / $x : A \Rightarrow \{(M, 0), (N, 0)\}$
- $M[x, y]\ M[y, x]$ / $x : A \Rightarrow \{(M, 0), (M, 1)\}$
- $M[x]\ y$ / $x : A, y : B$ fails because y occurs rigidly.

Iterated dependent product type $\Pi(\overrightarrow{x_i : A_i}).B = \Pi x_1 : A_1.\cdots.\Pi x_n : A_n.B$

Iterated dependent sum type $\Sigma(\overrightarrow{x_i : A_i}) = \Sigma x_1 : A_1.\cdots.\Sigma x_{n-1} : A_{n-1}.A_n$

$\boxed{\mathrm{Hoist}(\Delta)}$ Set of telescopes obtained by bringing an element of Δ to the front and possibly applying some pruning.

$$\mathrm{Hoist}(\overrightarrow{x_i : A_i}) = \{\mathrm{prune}(P), (x_j : \mathrm{prune}(P)(A_j), x_1 : A_1, \cdots, x_{j-1} : A_{j-1}, x_{j+1} : A_{j+1}, \cdots, x_n : A_n) \mid \exists P.A_j \mathbin{/} (x_1 : A_1, \cdots, x_{j-1} : A_{j-1}) \Rightarrow P\}$$

$$\Theta \mid \Gamma \vdash \mathrm{Id}_A(t_1, t_2) \stackrel{?}{\cong} \mathrm{Id}_B(u_1, u_2) \longmapsto \{\Theta \mid \Gamma \vdash A \stackrel{?}{=} B,\ \Theta \mid \Gamma \vdash t_2 \stackrel{?}{=} u_1,\ \Theta \mid \Gamma \vdash t_1 \stackrel{?}{=} u_2\}$$

$$\Theta \mid \Gamma \vdash \Pi\Delta.A \stackrel{?}{\cong} \Pi x{:}B_1.B \overset{\theta}{\longmapsto} \{\Theta \mid \Gamma \vdash A_j \stackrel{?}{\cong} B_1,\ \Theta \mid \Gamma, x_j \vdash \Pi\Delta'.A \stackrel{?}{\cong} B[x \mapsto x_j]\} \quad \text{if } \|\Delta\| > 1 \wedge \exists(\theta, (x_j : A_j, \Delta')) \in \mathrm{Hoist}(\Delta) \wedge x_j \notin \mathrm{FV}(\Pi\Delta'.A)$$

$$\Theta \mid \Gamma \vdash \Pi\Delta.A \stackrel{?}{\cong} \Pi x{:}B_1.B \overset{\theta}{\longmapsto} \{\Theta \mid \Gamma \vdash A_j \stackrel{?}{=} B_1,\ \Theta \mid \Gamma, x_j \vdash \Pi\Delta'.A \stackrel{?}{\cong} B[x \mapsto x_j]\} \quad \text{if } \|\Delta\| > 1 \wedge \exists(\theta, (x_j : A_j, \Delta')) \in \mathrm{Hoist}(\Delta) \wedge x_j \in \mathrm{FV}(\Pi\Delta'.A)$$

$$\Theta \mid \Gamma \vdash \Sigma\Delta \stackrel{?}{\cong} \Sigma x{:}B_1.B_2 \overset{\theta}{\longmapsto} \{\Theta \mid \Gamma \vdash A_j \stackrel{?}{\cong} B_1,\ \Theta \mid \Gamma, x_j \vdash \Sigma\Delta' \stackrel{?}{\cong} B_2[x \mapsto x_j]\} \quad \text{if } \|\Delta\| > 1 \wedge \exists(\theta, (x_j : A_j, \Delta')) \in \mathrm{Hoist}(\Delta) \wedge x_j \notin \mathrm{FV}(\Sigma\Delta')$$

$$\Theta \mid \Gamma \vdash \Sigma\Delta \stackrel{?}{\cong} \Sigma x{:}B_1.B_2 \overset{\theta}{\longmapsto} \{\Theta \mid \Gamma \vdash A_j \stackrel{?}{=} B_1,\ \Theta \mid \Gamma, x_j \vdash \Sigma\Delta' \stackrel{?}{\cong} B_2[x \mapsto x_j]\} \quad \text{if } \|\Delta\| > 1 \wedge \exists(\theta, (x_j : A_j, \Delta')) \in \mathrm{Hoist}(\Delta) \wedge x_j \in \mathrm{FV}(\Sigma\Delta')$$

Figure 6. The **permute-decompose** rules and auxiliary definitions.

Telescope $\Delta ::= \overrightarrow{x_i : A_i}$ Telescope Length $\|\Delta\|$

$\boxed{t \mathbin{/} \Delta \Rightarrow P}$ Metavariable parameters P needs to be pruned to eliminate variables in Δ from t.

$$\frac{}{M[\overrightarrow{t}] \mathbin{/} \Delta \Rightarrow \{(M, i) \mid t_i \in \overrightarrow{t}, x \in \Delta, x \in \mathrm{FV}(t_i)\}} \qquad \frac{x \notin \Delta}{x \mathbin{/} \Delta \Rightarrow \emptyset} \qquad \frac{c : A := t \in \mathcal{S}}{c \mathbin{/} \Delta \Rightarrow \emptyset} \qquad \frac{}{\mathrm{Type} \mathbin{/} \Delta \Rightarrow \emptyset}$$

$$\frac{A \mathbin{/} \Delta \Rightarrow P \quad B \mathbin{/} \Delta \Rightarrow Q}{\Pi x{:}A.B \mathbin{/} \Delta \Rightarrow P \cup Q} \qquad \frac{t \mathbin{/} \Delta \Rightarrow P}{\lambda x.t \mathbin{/} \Delta \Rightarrow P} \qquad \frac{t \mathbin{/} \Delta \Rightarrow P \quad u \mathbin{/} \Delta \Rightarrow Q}{t\,u \mathbin{/} \Delta \Rightarrow P \cup Q}$$

$$\frac{A \mathbin{/} \Delta \Rightarrow P \quad B \mathbin{/} \Delta \Rightarrow Q}{\Sigma x{:}A.B \mathbin{/} \Delta \Rightarrow P \cup Q} \qquad \frac{t \mathbin{/} \Delta \Rightarrow P \quad u \mathbin{/} \Delta \Rightarrow Q}{(t, u) \mathbin{/} \Delta \Rightarrow P \cup Q} \qquad \frac{t \mathbin{/} \Delta \Rightarrow P}{\pi_1(t) \mathbin{/} \Delta \Rightarrow P} \qquad \frac{t \mathbin{/} \Delta \Rightarrow P}{\pi_2(t) \mathbin{/} \Delta \Rightarrow P}$$

$$\frac{}{\mathrm{Unit} \mathbin{/} \Delta \Rightarrow \emptyset} \qquad \frac{}{\mathrm{tt} \mathbin{/} \Delta \Rightarrow \emptyset} \qquad \frac{A \mathbin{/} \Delta \Rightarrow P \quad t \mathbin{/} \Delta \Rightarrow Q \quad u \mathbin{/} \Delta \Rightarrow R}{\mathrm{Id}_A(t, u) \mathbin{/} \Delta \Rightarrow P \cup Q \cup R}$$

$\boxed{\mathrm{prune}(P)}$ Metavariable substitution that prunes away metavariable parameters in P.

$$\mathrm{prune}(P) = [M[x_1, \cdots, x_n] \mapsto M'[x_{j_1}, \cdots, x_{j_{n-k}}], \cdots]$$

where M' is a fresh metavariable, $k = |\{i \mid (M, i) \in P\}|$, and $1 \le j_1 < \cdots < j_{n-k} \le n$ satisfy $(M, j_1), \cdots, (M, j_{n-k}) \notin P$

Figure 7. Pruning and auxiliary definitions.

This judgment does not handle nested metavariables. For instance, when eliminating x from $M[L[x], N[x]]$, the judgment only returns the pruning $\{(M, 0), (M, 1)\}$, although pruning the arguments of L and N individually would also eliminate the dependency. We currently leave such nested metavariables unhandled, as it is unclear whether this is useful in practice.

4.6 Guessing Metavariables

When a metavariable appears at a certain position of a term, we may partially guess its structure to make progress (the **guess** rule, Figure 8). For example, if a meta-application $M[\overrightarrow{t}]$ is applied to another term u, i.e. $M[\overrightarrow{t}]\,u$, we can infer that the result of the meta-application is a function and so should be substituted like $[M[\overrightarrow{x}] \mapsto \lambda y.M'[\overrightarrow{x}, y]]$, where M' is a fresh metavariable.

$$\Theta \mid \Gamma \vdash \Sigma x{:}M[\overrightarrow{t}].B \stackrel{?}{\cong} A \quad \overset{[M[\overrightarrow{x}]\mapsto \mathrm{Unit}]}{\longmapsto} \quad \{\Theta \mid \Gamma \vdash \Sigma x{:}\mathrm{Unit}.B \stackrel{?}{\cong} A\}$$

$$\Theta \mid \Gamma \vdash \Sigma x{:}M[\overrightarrow{t}].B \stackrel{?}{\cong} A \quad \overset{[M[\overrightarrow{x}]\mapsto \Sigma y{:}M_1[\overrightarrow{x}].M_2[\overrightarrow{x},y]]}{\longmapsto} \quad \{\Theta, M_1 : [\overrightarrow{\star}]\star, M_2 : [\overrightarrow{\star},\star]\star \mid \Gamma \vdash \Sigma x{:}(\Sigma y{:}M_1[\overrightarrow{t}].M_2[\overrightarrow{t},y]).B \stackrel{?}{\cong} A\}$$

if A is a dependent sum type or headed by a (parametrised) metavariable

$$\Theta \mid \Gamma \vdash \Sigma x{:}A.M[\overrightarrow{t}] \stackrel{?}{\cong} B \quad \overset{[M[\overrightarrow{x}]\mapsto \mathrm{Unit}]}{\longmapsto} \quad \{\Theta \mid \Gamma \vdash A \times \mathrm{Unit} \stackrel{?}{\cong} B\}$$

$$\Theta \mid \Gamma \vdash \Sigma x{:}A.M[\overrightarrow{t}] \stackrel{?}{\cong} B \quad \overset{[M[\overrightarrow{x}]\mapsto \Sigma y{:}M_1[\overrightarrow{x}].M_2[\overrightarrow{x},y]]}{\longmapsto} \quad \{\Theta, M_1 : [\overrightarrow{\star}]\star, M_2 : [\overrightarrow{\star},\star]\star \mid \Gamma \vdash \Sigma x{:}A.\Sigma y{:}M_1[\overrightarrow{t}].M_2[\overrightarrow{t},y] \stackrel{?}{\cong} B\}$$

if B is a dependent sum type or headed by a (parametrised) metavariable

$$\Theta \mid \Gamma \vdash \Pi x{:}M[\overrightarrow{t}].B \stackrel{?}{\cong} A \quad \overset{[M[\overrightarrow{x}]\mapsto \mathrm{Unit}]}{\longmapsto} \quad \{\Theta \mid \Gamma \vdash \Pi x{:}\mathrm{Unit}.B \stackrel{?}{\cong} A\}$$

$$\Theta \mid \Gamma \vdash \Pi x{:}M[\overrightarrow{t}].B \stackrel{?}{\cong} A \quad \overset{[M[\overrightarrow{x}]\mapsto \Sigma y{:}M_1[\overrightarrow{x}].M_2[\overrightarrow{x},y]]}{\longmapsto} \quad \{\Theta, M_1 : [\overrightarrow{\star}]\star, M_2 : [\overrightarrow{\star},\star]\star \mid \Gamma \vdash \Pi x{:}(\Sigma y{:}M_1[\overrightarrow{t}].M_2[\overrightarrow{t},y]).B \stackrel{?}{\cong} A\}$$

if A is a dependent product type or headed by a (parametrised) metavariable

$$\Theta \mid \Gamma \vdash \Pi x{:}A.M[\overrightarrow{t}] \stackrel{?}{\cong} B \quad \overset{[M[\overrightarrow{x}]\mapsto \Pi y{:}M_1[\overrightarrow{x}].M_2[\overrightarrow{x},y]]}{\longmapsto} \quad \{\Theta, M_1 : [\overrightarrow{\star}]\star, M_2 : [\overrightarrow{\star},\star]\star \mid \Gamma \vdash \Pi x{:}A.(\Pi y{:}M_1[\overrightarrow{t}].M_2[\overrightarrow{t},y]) \stackrel{?}{\cong} A\}$$

if A is a dependent product type or headed by a (parametrised) metavariable

Figure 8. The **guess** rules for dependent product/sum types. May unblock the **normalise** or **permute-decompose** rule. The other transitions rules for application and projection are omitted.

It is interesting that we can guess the structure of a meta-application if it appears in the (co)domain of a dependent product or dependent sum type. In the rewrite system R, these types "eliminate" dependent sums and units at their domain. Therefore, it is a valid guess that the result of a meta-application at the position is a dependent sum or unit type. That is, inferring $[M[] \mapsto \Sigma y{:}M_1[].M_2[x]]$ of $\Pi x{:}M[].A$, for instance. This induces further normalisation by the **normalise** rule and may contribute to finding a unifier, as shown in Figure 9. We would not have been able to solve the constraint without the guess in this particular example. Similarly, in order to give the **permute-decompose** rule a chance to fire, we can infer that the result of a metavariable application at the codomain of a dependent product/sum type is again dependent product/sum type, respectively.

Applying such guesses blindly can sometimes only make a constraint larger without contributing to a solution. For example, consider the constraint $(M[] \to \mathrm{Type}) \stackrel{?}{\cong} \mathrm{Unit}$ (context omitted). Guessing $[M[] \mapsto \Sigma x{:}M_1[].M_2[x]]$ produces the larger constraint $(\Pi x{:}M_1[].M_2[x] \to \mathrm{Type}) \stackrel{?}{\cong} \mathrm{Unit}$. This does not help in finding a solution. The issue here is that we did not take into account the Unit on the right-hand side, which cannot be obtained from such a guess. Therefore, we impose certain conditions on some of the **guess** rules to eliminate such unproductive guesses. For example, we only guess $[M[] \mapsto \Sigma y{:}M_1[].M_2[x]]$ for $\Pi x{:}M[].A$ when the other side of the constraint is a dependent product type or is headed by a metavariable application.

4.7 Solving Metavariables

Flex-Rigid constraints are solved similarly to [14]. To solve a flex-rigid constraint, i.e. $\Theta \mid \Gamma \vdash M[t_1, \cdots, t_n] \stackrel{?}{\cong} u$ or $\Theta \mid \Gamma \vdash M[t_1, \cdots, t_n] \stackrel{?}{=} u$, the algorithm goes through a set of candidate solutions, which is generated using a heuristic that resembles Huet-styles projection bindings generalised for untyped SOAS.

5 Implementation and Examples

We have implemented a prototype of our unification algorithm in Haskell[2]. In this section we will describe the implementation and give concrete examples.

5.1 Implementation Details

There are two differences in the core language in the implementation and one described in Section 3:

- Contrary to the assumption made in Section 3, the core language has the inductively defined identity type, whose swap operation is not definitionally involutive.
- The core language has both non-dependent and dependent function and pair types. This sometimes allows skipping dependency checking, thus enabling efficient determination of whether the constraint should be solved up to the isomorphism or definitional equality on decomposition. However, dependency checking is still needed when permuting function domains and pair components. This issue could be addressed by

[2]GitHub repository: https://github.com/wasabi315/dependent-type-search

$$\{\Theta \mid \Gamma \vdash M[] \to A \stackrel{?}{\cong} \Pi B{:}\mathrm{Type}.B \to A\} \text{ where } \Theta \mid \Gamma = (M\colon []\star \mid A\colon \star)$$

$$\overset{[M[]\mapsto \Sigma x{:}M_1[].M_2[x]]}{\longmapsto} \{\Theta' \mid \Gamma \vdash (\Sigma x{:}M_1[].M_2[x]) \to A \stackrel{?}{\cong} \Pi B{:}\mathrm{Type}.B \to A\} \text{ where } \Theta' = M\colon []\star, M_1\colon []\star, M_2\colon [\star]\star \quad \textbf{(guess)}$$

$$\longmapsto \{\Theta' \mid \Gamma \vdash \Pi x{:}M_1[].M_2[x] \to A \stackrel{?}{\cong} \Pi B{:}\mathrm{Type}.B \to A\} \quad \textbf{(normalise)}$$

$$\longmapsto \{\Theta' \mid \Gamma \vdash M_1[] \stackrel{?}{=} \mathrm{Type},\ \ \Theta' \mid \Gamma, B\colon \star \vdash M_2[B] \to A \stackrel{?}{\cong} B \to A\} \quad \textbf{(decompose)}$$

$$\overset{[M_1[]\mapsto \mathrm{Type}]}{\longmapsto} \{\Theta' \mid \Gamma, B\colon \star \vdash M_2[B] \to A \stackrel{?}{\cong} B \to A\}$$

$$\longmapsto \{\Theta' \mid \Gamma, B\colon \star \vdash M_2[B] \stackrel{?}{\cong} B,\ \ \Theta' \mid \Gamma, B\colon \star \vdash A \stackrel{?}{\cong} A\} \quad \textbf{(decompose)}$$

$$\overset{[M_2[x]\mapsto x]}{\longmapsto} \{\Theta' \mid \Gamma, B\colon \star \vdash A \stackrel{?}{\cong} A\} \longmapsto \emptyset \quad \textbf{(decompose)}$$

Figure 9. Guessing the structure of a metavariable at the argument position

carrying extra information about variable usage at binding site, in a similar way to [22].

Definitions are unfolded only when necessary, following the technique described in [12]. This approach avoids unnecessary unfolding and enables clearer presentation of terms in substitutions for metavariables in queries.

5.2 Examples

The implementation features a prototype interactive search tool for the core language. It performs a straightforward search over the list of signatures from modules specified, attempting to unify each signature with the query. To support search up to generalisation, top-level dependent function types in signatures are instantiated non-deterministically with fresh metavariables. Type-checking of unifiers is not yet implemented, so search results may include ill-typed solutions. We have prepared a small dataset comprising 4 modules and a total of 38 constant definitions. All search examples shown below completed instantly. We plan to evaluate the performance of the search tool on larger, real-world datasets in the future.

One may want to search for the list recursor with the query in the first line:

```
> (A B : Type) -> (B * A -> B) -> List A -> B -> B
List.foldr :
  (A B : Type) -> (A -> B -> B) -> B -> List A -> B
  instantiation: {A = A, B = B}
```

The search tool successfully identifies `foldr` from the query because the search operates modulo domain reordering and currying. It not only displays the matched library components, but also shows how they are instantiated and which terms are assigned to the metavariables in the query. In this particular example, the instantiation is straightforward.

The tool also searches modulo symmetry of identity types.

```
> (m : Nat) -> Eq Nat (suc m) 0 -> False
Natural.zero-suc-neq :
  (m : Nat) -> Not (Eq Nat 0 (suc m))
  instantiation: {m = m}
```

The search is modulo δ-conversion and allows the user to include (parametrised) metavariables in the query. Note that the definition of addition is not expanded in the shown metavariable substitution, thanks to lazy unfolding of definitions.

```
> (m n : Nat) -> Eq Nat ?F[m,n] ?F[n,m]
Natural.add-comm : Commutativity Nat add
  instantiation: {?F[x0,x1] = add x0 x1}
```

The example below shows the flexibility of search up to generalisation (and δ-conversion). The commutativity statement got successfully instantiated with 42.

```
> (m : Nat) -> Eq Nat (add 42 m) (add m 42)
Natural.add-comm : Equality.Commutativity Nat add
  instantiation: {x = 42, y = m}
```

This last example shows the combination of search up to generalisation and more premises (through the use of the metavariable manually added).

```
> (A : Type) -> (A -> A -> A) -> List A -> ?M[A] ->
    A
List.foldr :
  (A B : Type) -> (A -> B -> B) -> B -> List A -> B
  instantiation: {A = A, B = A}
  substitution: {?M[x0] = x0}

Equality.transport :
  (A B : Type) -> Eq Type A B -> A -> B
  instantiation: {A = A -> A -> A, B = List A -> A}
  substitution: {?M[x0] = Eq Type (x0 -> x0 -> x0)
    (List x0 -> x0)}

Basic.id : (A : Type) -> A -> A
  instantiation: {A = (A -> A -> A) -> List A -> A}
  substitution: {?M[x0] = (x0 -> x0 -> x0) -> List
    x0 -> x0}
... (omitted)
```

The query resulted in many suggestions including not only foldr, as expected, but also highly non-trivial ones. For better display, we could rank (or even filter) the search results depending on the complexity of instantiation and substitution, as described in [21].

6 Conclusion and Future Work

In this work, we developed a semi-algorithm for unification modulo type isomorphisms, with the goal of enabling more flexible type-based library search than previous tools.

Several aspects remain open for exploration. First, as discussed in Section 4, we intend to incorporate type information from libraries and ensure well-typedness throughout unification. To achieve this, we have to track and verify the types of metavariables, since some transition rules may otherwise produce ill-typed solutions. For example,it is necessary to check whether a metavariable remains well-typed after removing pruned parameters [1]. Second, our algorithm should be extended to actually compute isomorphisms. Currently, the algorithm only determines whether type isomorphisms exist. Producing concrete isomorphisms would provide witnesses to help users understand why library components match, and could even assist with goal completion (i.e., code generation) when integrated into ITPs. This extension is also necessary for lifting domain restrictions, as it would require handling "dependent" isomorphisms (those that depend on other isomorphisms). Third, we should consider support for other pervasive language features, including implicit arguments, type classes (instance arguments), and inductive definitions. For type classes, it would be valuable to develop a more principled approach to extending methods, for example, by employing order-sorted unification as described by [16].

Acknowledgments

The authors would like to thank the anonymous reviewers for their feedback on the draft of this paper. This work was supported by JST SPRING, Japan Grant Number JPMJSP2180, and in part by JSPS KAKENHI Grant Numbers JP22K11967 and JP24K14892.

References

[1] Andreas Abel and Brigitte Pientka. 2011. Higher-order dynamic pattern unification for dependent types and records. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 10–26.

[2] Clément Allain, Gabriel Radanne, and Laure Gonnord. 2021. Isomorphisms are back!. In *ML 2021 - ML Workshop*. Virtual, France, 1–3. https://hal.science/hal-03355381

[3] Joachim Breitner. 2023. Loogle. https://loogle.lean-lang.org/

[4] Claudio Sacerdoti Coen. 2024. An Indexer and Query Language for Libraries written in LambdaPi/Dedukti. https://europroofnet.github.io/_pages/WG4/Tbilisi24/sacerdoti.pdf

[5] David Delahaye. 2000. Information retrieval in a coq proof library using type isomorphisms. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 131–147.

[6] Roberto Di Cosmo. 2005. A short survey of isomorphisms of types. *Math. Struct. Comput. Sci.* 15, 05 (Oct. 2005), 825.

[7] Roberto DiCosmo. 1992. Type isomorphisms in a type-assignment framework. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '92*. ACM Press, New York, New York, USA.

[8] Roberto DiCosmo. 2012. *Isomorphisms of Types: from λ-calculus to information retrieval and language design* (1995 ed.). Birkhäuser, Cambridge, MA.

[9] Marc Etter and Farhad Mehta. 2024. Towards type-directed API search for mainstream languages. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Type-Driven Development*. ACM, New York, NY, USA, 50–61.

[10] Marcelo Fiore and Chung-Kil Hur. 2010. Second-order equational logic (extended abstract). In *Computer Science Logic*. Springer Berlin Heidelberg, Berlin, Heidelberg, 320–335.

[11] Idris. 2025. Documentation for the Idris Language. https://docs.idris-lang.org/en/latest/

[12] András Kovács. 2024. Efficient Evaluation with Controlled Definition Unfolding. Paper presentation. https://andraskovacs.github.io/pdfs/wits24prez.pdf 3rd Workshop on the Implementation of Type Systems (London, United Kingdom, 20 January 2024).

[13] Nikolai Kudasov. 2023. E-unification for second-order abstract syntax. 10:1–10:22 pages.

[14] Nikolai Kudasov. 2025. Free monads, intrinsic scoping, and higher-order preunification. In *Lecture Notes in Computer Science*. Springer Nature Switzerland, Cham, 22–54.

[15] Lambdapi. 2025. Lambdapi User Manual. https://lambdapi.readthedocs.io/en/latest

[16] Brian Matthews. 1992. Reusing functional code using type classes for library search. In *ERCIM Workshop on Software Reuse, Heraklion, Crete*.

[17] Neil Mitchell. 2004. Hoogle. https://hoogle.haskell.org/

[18] Neil Mitchell. 2011. Hoogle: Finding Functions from Types. https://ndmitchell.com/downloads/slides-hoogle_finding_functions_from_types-16_may_2011.pdf Presentation from TFP 2011.

[19] P Narendran, F Pfenning, and R Statman. 2002. On the unification problem for Cartesian closed categories. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*. IEEE Comput. Soc. Press, 57–63.

[20] Mikael Rittri. 1989. Using types as search keys in function libraries. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 174–183.

[21] M Rittri. 1993. Retrieving library functions by unifying types modulo linear isomorphism. *Theor. Inform. Appl.* 27, 6 (1993), 523–540.

[22] Masahiko Sato, Randy Pollack, Helmut Schwichtenberg, and Takafumi Sakurai. 2013. Viewing λ-terms through maps. *Indag. Math. (N.S.)* 24, 4 (Nov. 2013), 1073–1104.

[23] Stephan Schulz. 2013. Simple and efficient clause subsumption with feature vector indexing. In *Automated Reasoning and Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 45–67.

[24] Ben Sherman. 2014. Type-directed search with dependent types. https://haskelldc.github.io/type-search-8-12-14/slides/type-search.pdf

[25] The Rocq Development Team. 2025. The Rocq Reference Manual – Release 9.0.0. https://rocq-prover.org/doc/v9.0/refman/

[26] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.* 3, ICFP (July 2019), 1–29.

[27] Lukas Wegmann, Farhad Mehta, Peter Sommerlad, and Mirko Stocker. 2016. Scaps: type-directed API search for Scala. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. ACM, New York, NY, USA.

Received 2025-06-22; accepted 2025-07-23