Light-speed type unification modulo isomorphisms

Emmanuel Arrighi, Gabriel Radanne

Light-speed type unification modulo isomorphisms

Emmanuel Arrighi
EnsL, UCBL, CNRS, LIP
emmanuel.arrighi@ens-lyon.fr
Gabriel Radanne
Inria, EnsL, UCBL, CNRS, LIP
gabriel.radanne@inria.fr

Abstract

One essential task of programmers it to use the right library and function. But what is the *right* function? Most developers use their experience and knowledge of the ecosystem to intuit where to look for. Tools like Sherlodoc or Hoogle provide *textual search* in an ecosystem of functions, easing the quest of the developer.

To go further, Dowsing allow searching OCaml functions by their *types*: the developer provides a type, and Dowsing finds a function whose type is suitable. More precisely, it finds any functions whose type is more general, up to a set of isomorphisms such as currying or reordering of arguments. Results are guaranteed to be sound (the type does match) and complete (all such functions are found).

Unfortunately such search is costly, relying on "unification modulo isomorphisms" (which is NP-complete). So far, Dowsing relied on database-style indexation techniques and state-of-the-art unification algorithms, enabling search over a set of local libraries, but exhibiting poor performances for very polymorphic queries.

In this article, we present several improvements to unification modulo isomorphism which makes Dowsing up to *500 times faster* on complex queries. This brings Dowsing in a new area where it can be used interactively by developers.

ACM Reference Format:
Emmanuel Arrighi, EnsL, UCBL, CNRS, LIP, emmanuel.arrighi@ens-lyon.fr and Gabriel Radanne, Inria, EnsL, UCBL, CNRS, LIP, gabriel.radanne@inria.fr . 2024. Light-speed type unification modulo isomorphisms. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

When coding, libraries are great tools to avoid having to reimplement the wheel over and over. Especially for intricate functions if someone did it once, we don't want to redo the job, therefore, we would like to be able to find functions if they exist. But answering the question "Is there a function that does X?" is not trivial. Consider a function summing a list of integers. We can first look into the documentation of the `List` module. This requires some existing knowledge, and might not be fruitful. For instance, it could be in the `List` or in the `Int` module.

Existing tools, such as Hoogle[1] for Haskell or Sherlocode[2] for OCaml, allow using a textual search for functions, their types and documentation. While such tools scale very well and provide some result, they are, by essence, incomplete: they will never fine the `fold_left` function which allow you to easily implement sum, since it relies on polymorphism.

Ideally, we want search by *functionality*. While difficult in general, **DBLP:journals/jfp/Rittri91** proposed an approximation by using the expected type of a function (consisting of the type of the arguments and the return type). To allow searching for functions that are more abstract than the query, he proposed to base the search for functions using the matching relation. For example, if the query is `int list -> int`, the search algorithm will include `List.length:'a list -> int` and `List.hd: 'a list -> 'a` in the result. To avoid being strict with the order of the arguments, or the way they are passed to the function, Rittri proposed to augment the matching relation with some type isomorphisms. By including type isomorphisms, the query `'a list * 'a -> bool` can be used to find `List.mem:'a -> 'a list -> bool`. The main drawback of this approach is that the problem of matching modulo isomorphisms is a computationally hard for the polymorphic lambda calculus [**DBLP:journals/jar/KapurN92**; **DBLP:journals/jsyml/NarendranPS97**; **solov1983category**] and even quickly becomes undecidable for richer type systems. Rittri implemented his approach for the standard library of Lazy ML, which has around 200 functions in 1991. Even though his approach worked well back then, it does not easily scale to larger ecosystems such as the full OCaml ecosystem. Notably, Opam, the package manager for OCaml,

[1] https://hoogle.haskell.org
[2] https://doc.sherlocode.com/

Emmanuel Arrighi
EnsL, UCBL, CNRS, LIP
emmanuel.arrighi@ens-lyon.fr and Gabriel Radanne
Inria, EnsL, UCBL, CNRS, LIP
gabriel.radanne@inria.fr

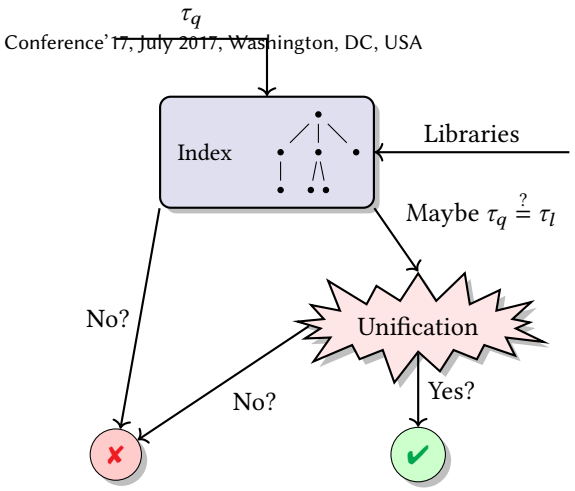

Figure 1. Architecture of *Dowsing*. The library contains a set of *library types* τ_l which forms an index. The developers provide a *query type* τ_q to find.

contains more than 4000 packages, each of which has between 10 a 100 functions.

Dowsing [**allain2021isomorphisms**] is a tool based on [**DBLP:journals/jfp/Rittri91**]'s approach scaling to medium size collections of libraries for the OCaml ecosystem. Up until now, *Dowsing* was able to scale to a medium ecosystem, such as the set of locally installed Opam packages. For simple queries, performance would be reasonable (less than 10s), but would degrade for very polymorphic queries (up to several minutes).

In this talk, we will present improvements to unification modulo isomorphism which provide a *500 times speedup* over previous state-of-the-art algorithms for polymorphic queries, bringing all queries under the 10s range. We briefly present *Dowsing*'s architecture and illustrate the ideas behind these improvements below.

2 Architecture of *Dowsing*

The goal of *Dowsing* is, given an ecosystem of functions, find the ones whose types unify modulo isomorphisms. For this purpose, the architecture of *Dowsing*, presented in Fig. 1, is centered around two main components: a database of functions *indexed by types*, and a 1-to-1 unification modulo isomorphisms decision procedure.

Fast Indexing of Types. *Dowsing*'s index is described by [**allain2021isomorphisms**]. It leverages ideas similar to a database: given a query type τ_q, we want to find a number of library types τ_l which *may* unify. We will then confront these candidates types with a real unification modulo isomorphisms procedure. At this stage, the goal is thus to reject as many non-unifiable types as possible, as efficiently as possible, in order to avoid calling the expensive unification algorithm.

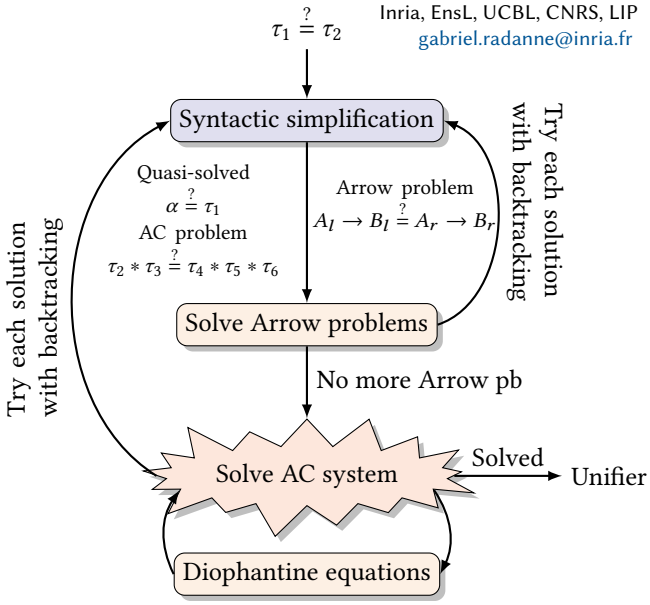

Figure 2. Unification Algorithm by **DBLP:journals/jar/Boudet93**.

Our first technique is to collect *features* of types equipped with a compatibility relation such that given τ_q and a τ_l may unify only if their features are compatible. This means features act as a filter to eliminate candidates. Features can be precomputed on the database of types and arranged in a data-structures (in our case, a Trie), providing easy search.

Furthermore, *Dowsing* leverages the "more general" relation: indeed given two library types τ_l and τ'_l, if τ_l is more general than τ'_l and if τ_q doesn't unify with τ_l, then it can't unify with τ'_l. The "more general" relation in ML type systems form a lattice which can be looked up efficiently.

In practice, as we will see in the experimental section, these two techniques provide excellent scaling for medium to large database on queries with little polymorphism, often filtering all but a handful of candidates. Still we need to verify these candidates actually unify with the query.

Unification modulo Isomorphisms. Unification modulo an equational theory is well studied [**DBLP:journals/mscs/Cosmo05**; **DBLP:conf/cade/SiekmannS82**]. Here we consider three isomorphisms on simple types: Associativity and Commutativity (AC) of tuples, and currying of arrows. We purposefully eschew so-called "non-linear" isomorphisms such as distributivity of arrow over tuples, as they quickly lead to undecidability [**DBLP:journals/jsyml/NarendranPS97**]. Conveniently, algorithms for unification modulo Associativity and Commutativity have been the subject of much attention via a public contest [**DBLP:journals/jar/BurckertHKSSTZ88**], yielding a very efficient algorithm by **DBLP:journals/jar/Boudet93**. This algorithm can easily be extended to Currying. We illustrate the extended procedure in Fig. 2 and quickly summarize it now.

DBLP:journals/jar/Boudet93's algorithm works as follows. Given two types τ_1 and τ_2, there a first step of syntactic simplification which looks at the structure of τ_1 and τ_2 and separate the initial problem into three categories of equations: "Quasi-solved" equations of the form $\alpha \stackrel{?}{=} \tau$; "Tuple problems" of the form $\tau * \ldots \stackrel{?}{=} \tau * \ldots$; and "Arrow problems" of the form $A_l \to B_l \stackrel{?}{=} A_r \to B_r$. Once this phase is done, we obtain an Arrow system and an Tuple system to solve. Each system contain an "elementary" problem relating to only one theory (arrows, or tuples). It simply remains to solve these elementary problems and explore all solutions with a backtracking algorithm.

We first solve the Arrow problems. Here, the key observation is to note that, the return type of each side, could or not be unified to an arrow type, leading to an additional argument to the function. This leaves us with the following four possible solutions for $A_l \to B_l \stackrel{?}{=} A_r \to B_r$:

1. $A_l \stackrel{?}{=} A_r \quad \wedge \quad B_l \stackrel{?}{=} B_r$
2. $A_l * \alpha_l \stackrel{?}{=} A_r \quad \wedge \quad B_l \stackrel{?}{=} \alpha_l \to B_r$
3. $A_l \stackrel{?}{=} A_r * \alpha_r \quad \wedge \quad \alpha_r \to B_l \stackrel{?}{=} B_r$
4. $A_l * \alpha_l \stackrel{?}{=} A_r * \alpha_r \quad \wedge \quad B_l \stackrel{?}{=} \alpha_l \to \beta \quad \wedge \quad \alpha_r \to \beta \stackrel{?}{=} B_r$

In Item 1, none of the result unify with an arrow type, in Item 2 and Item 3, only one side does and in Item 4 both side unify with an arrow type. For each of those solutions, we try them using backtracking. We add the equation in the current system and start over with the simplification step.

Once there are no more Arrow problems, we continue with solving the Tuple systems. It turns out that such system of equation with Associative-Commutative products can be encoded into a single system of linear diophantine equations [**DBLP:conf/lics/BoudetCD90**]. From the solutions of the linear diophantine equation, we can recompose a set of type unifiers, which are tried one by one using backtracking. The procedure restarts from the simplification step until the problem is fully solved or a contradiction is met.

3 Fast ACCurry-Unification

For, polymorphic queries, candidates are (predictably) more difficult to filter out, as establishing non-unification might require non-local reasoning. In this case, the index returns many candidates, thus requiring many calls to the decision procedure. In this case, the performance of the algorithm by **DBLP:journals/jar/Boudet93** is not sufficient.

As we have seen, solving core AC unification relies on a translation to diophantine equations. Solution of the diophantine equations are then translated back to type unifiers. This last translation is precisely where the main combinatory explosion happen. Let us consider the types $\alpha * \texttt{int list} \to \texttt{unit}$ and $c_1 * c_2 * \ldots * c_{14} \to \beta$. After the initial phase, it yields a diophantine system with 30 solutions, which leads to 49150 type unifiers! Most of those are in fact irrelevant: they do not account for the fact that `int list` is present in the initial type, and will be promptly rejected later. To solve this problem we introduce the following improvements.

A simpler system decomposition. The encoding proposed by **DBLP:journals/jar/Boudet93** produces a single big system, and then filters out the solutions that will not give a valid solution for the unification modulo AC problems. The filtering is based on two remarks: 1. two different constants cannot be unified, (an `int` cannot be unified with a `float`), 2. a constant cannot be unified with a tuple.

This "a posteriori" filtering is valid but makes for awkward reasoning. Furthermore, we would prefer to avoid producing a big system (solving diophantine equations is still, after all, an NP-hard problem). Therefore, we instead chose to encode the AC system into several smaller systems of diophantine equations which prevent wrong solutions "a priori". The idea of the split is the following: given a system of AC problems we remark that a solution to the system can be seen as a substitution such that after applying it, each type appears the same number of time on both sides of each equations. Therefore, as `int` cannot be unified with `float`, it is not useful to consider `int` and `float` at the same time. It turns out that the other way around is also valid, in the sense that given for each constant c a partial solution that satisfies each equation when looking only at c, we can build a solution for the full system. Therefore, we create a system of linear diophantine equations per constant and one for the rest. By doing this, we obtain the same solution as [**DBLP:journals/jar/Boudet93**], but with smaller system of diophantine equations. This further simplifying encoding and decoding of the problem, unlocking our next improvement.

Shape classification. The filtering of **DBLP:journals/jar/Boudet93** and the decomposition in several systems that we do is based on the idea that we know, statically, that some pairs of types cannot be unified. Therefore, we can already separate them. It is obviously the case for different constants, as is done originally, but it can be extended to other types. For instance, we know that a `'a list` cannot be unified with a `float`. We thus introduce the notion of *shapes* as a necessary criterion for non-unification. Then, given a notion of shapes, we can partition the types of an AC system into several sets of types plus a set of variables with the property that for two non variable types, if they belong to different set then they cannot be unified. As a first implementation, we use the head symbol of a type as the shape: types of the form `_ list` and `_ result` will be separated, but `int list` and `float list` will not. Based on this partition in several bags of similar shapes, we apply the same idea as before: splitting the AC system into even more systems of diophantine equations. This means that we obtain very small sets of solutions to be combined, and avoid an "a posteriori" filtering.

Emmanuel Arrighi
EnsL, UCBL, CNRS, LIP
emmanuel.arrighi@ens-lyon.fr and Gabriel Radanne
Inria, EnsL, UCBL, CNRS, LIP
gabriel.radanne@inria.fr

Query	Full	Idx	Full+shape	Idx+shape
`int` → `int` → `int`	6.9s	0.02s	6.9s	0.02s
α → `int` → `unit`	17.3s	1.9s	7.1s	1.1s
α → α `list` → `bool`	11.6s	3.6s	8.5s	1.1s
$(\alpha * \beta) \to (\alpha \to \gamma) \to (\beta \to \delta) \to (\gamma * \delta)$	18.9m	18.3m	**12.1s**	**2.5s**

Figure 3. Experimental Results for a database of 50175 functions. Measured on an AMD Ryzen 7 2700U, 16Go de RAM.for a database of types notably the OCaml stdlib, `containers` and `base`.

4 Early Experimental Results

Early results are shown in Fig. 3. We measured the time taken by four queries of increasing complexity on consumer hardware. For these four queries, we report: (`Full`) a complete scan of the database without using the index, without shapes; (`Idx`) Regular search which uses the index, without shapes; (`Full+shape`) and (`Idx+shape`) the same, with our recent addition of shapes. We can observe three things: 1. On monomorphic queries, Indexing is already very efficient. Shapes provide no improvements 2. On slighly polymorphic queries, Indexing and Shapes combine very well and yield decent results 3. On highly polymorphic queries, Indexing does nothing, but Shapes make intractable queries trivial.

5 Work in Progress and Conclusion

There are still many leads to improve *Dowsing* further. First, we noted that many unifiers are symmetrical. Exploiting such symmetries would allow cutting the search by a factor of 2 on many queries. Additionally, we currently do not use neutrality of the `unit` element over products. Leveraging this axiom could provide better search, but also simplify unifiers, yielding better performances. We also hope to extend the expressivity of our system, which so far doesn't handle the more arcane features of OCaml (such as polymorphic variants).

Thanks to our improvements, *Dowsing* is now efficient enough that it can be integrated into common usage by developers. We are currently in the process of integrating *Dowsing* with Sherlodoc and odoc, to make it easy to use.

In the presentation, we will present our tool in practice, explain our recent improvements in more detail and illustrate it on concrete examples.